Static Data Flow Analysis for Android Applications

Statische Datenflussanalyse für Android-Anwendungen
Vom Fachbereich Informatik der Technische Universität Darmstadt
zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)
genehmigte Dissertation von Steven Arzt, M.Sc. M.Sc. aus Heidelberg

Tag der Einreichung: 16.11.2016
Tag der Prüfung: 22.12.2016

1. Gutachten: Prof. Eric Bodden, Ph.D.
2. Gutachten: Prof. Dr. Andreas Zeller
3. Gutachten: Prof. Dr. Patrick Eugster

Januar 2017 — Darmstadt — D 17

TECHNISCHE
UNIVERSITÄT
DARMSTADT

SECURE
SOFTWARE ENGINEERING
GROUP

CASED

EC SPRIDE

Static Data Flow Analysis for Android Applications
Statische Datenflussanalyse für Android-Anwendungen

Genehmigte Dissertation von Steven Arzt aus Heidelberg

1. Gutachten: Prof. Eric Bodden, Ph.D.
2. Gutachten: Prof. Dr. Andreas Zeller
3. Gutachten: Prof. Dr. Patrick Eugster

Tag der Einreichung: 16.11.2016
Tag der Prüfung: 22.12.2016

Darmstadt — D 17

Abstract

Mobile phones have become important daily companions for millions of people which help to organize both their private and their professional lives. Having access to data such as the calendar or the address book anywhere, anytime, has become commonplace. Sensor data such as the phone's GPS location and accelerometer help users navigate through the physical world. Users can furthermore extend the functionality of their phone using small programs called *apps* from various developers and vendors in an open ecosystem. Undoubtedly, having all this data merged on a device that is always-on and always-connected and that can easily be extended with new software greatly improves user convenience. On the other hand, it also poses new questions with regard to privacy and security. Apps may misuse the data stored on the phone or obtained from the sensors to infringe upon the user's privacy. In fact, companies already now use location data and app usage statistics to build user profiles for the purpose of targeted advertisement. The user is oftentimes unaware of these data leaks originating from his phone and has little means for analyzing the actual behavior of a given app with regard to privacy.

Static data flow analysis has been proposed as a means for automatically enumerating the data flows inside a program. Still, either do not support Android's platform-specific semantics or fall short on precision, recall, or scalability. In this thesis, we therefore propose techniques for efficiently and precisely performing static data flow analysis on real-world binary-only Android apps with large code sizes. We present the FlowDroid tool and show that it can detect data leaks in popular apps such as Facebook, Paypal, and LinkedIn. The FlowDroid reports improve the user's digital sovereignty by allowing his to asses the behavior of the app before installing it on his device and thereby entrusting it with his personal data. We allow the user to verify which of his data leaves the device and how. On the DroidBench micro-benchmark suite, we show that FlowDroid achieves a precision of more than 87% and a recall of over 84%, thereby outperforming state-of-the-art tools from academia and industry. Additionally, FlowDroid has already been used as a building-block for many other works in the field.

Zusammenfassung

Smartphones sind für Millionen von Menschen zu täglichen Begleitern geworden, mit den sie sowohl ihr privates als auch ihr geschäftliches Leben organisieren. An jedem Ort und zu jeder Zeit Zugriff auf Daten wie den Terminkalender oder das Adressbuch zu haben, gilt heute als selbstverständlich. Sensordaten wie die GPS-Position und der im Gerät verbaute Kompass helfen Benutzern, durch die reale Welt zu navigieren. Mit zusätzlichen Programmen, sogenannten *Apps*, können Benutzer die Funktionalität ihres Geräts zudem erweitern. Apps werden in einem offenen Ökosystem von verschiedensten Anbietern und Entwicklern vertrieben. Alle Daten an einem Ort zu haben, auf einem Gerät, das immer verfügbar und immer verbunden ist, bietet dem Benutzer zweifellos ein hohes Maß an Komfort. Auf der anderen Seite wirft es jedoch auch neue Fragen zu Datenschutz und Privatsphäre auf. Tatsächlich nutzen Firmen bereits heute Daten von mobilen Endgeräten, um Benutzerprofile für zielgruppenorientierte Werbung zu erzeugen. Der Endnutzer ist sich dieser Datenextraktion von seinem Gerät oftmals nicht bewusst und verfügt auch kaum über Möglichkeiten, sich darüber zu informieren, wie Apps mit seinen Daten tatsächlich umgehen.

Statische Datenflussanalyse ist eine Technik, mithilfe derer automatisiert Datenflüsse in Programmen erkannt werden können. Bisherigen Techniken fehlt jedoch entweder die Unterstützung für die spezifische Semantik der Android-Plattform oder sie liefern keine hinreichenden Ergebnisse in Beziug auf Genauigkeit, Vollständigkeit, Skalierbarkeit oder Geschwindigkeit. In dieser Doktorarbeit stellen wir daher Technken zur statischen Datenflussanalyse vor, mit denen reale Android-Apps mit großen Codemengen effizient analysiert werden können, auch wenn diese nur im Binärcode vorliegen. Wir präsentieren das FlowDroid-Werkzeug, welches präzise und weitestgehend vollständige Datenflüsse aus populären Apps wie Facebook, PayPal und LinkedIn extrahiert. Die Ergebnisberichte von FlowDroid verbessern die digitale Souveränität des Benutzers, da dieser sich nun selbst ein Bild davon machen kann, wie eine App mit seinen Daten umgeht, bevor er die App auf seinem Gerät installiert und ihr somit seine Daten anvertraut. Mit FlowDroid kann der Benutzer nachvollziehen, welche Daten von seinem Gerät erhoben und an Dritte gesendet werden. Auf der DroidBench Micro Benchmark-Suite liefert FlowDroid mehr als 87% aller erwarteten Ergebnisse. 84% aller gemeldeten Ergebnisse sind korrekt. Diese Werte liegen deutlich über den Ergebnissen der bisher aktuellen Analysewerkzeuge, sowohl aus dem akademischen als auch aus dem industriellen Kontext.

Acknowledgments

First, I would like to thank the organizations that employed me during the time of writing this thesis and conducting the research that lead up to it, namely TU Darmstadt and Fraunhofer SIT. Special thanks goes to my thesis supervisor Eric Bodden who always gave me great advice and motivation for my work as well as to Markus Schneider from Fraunhofer SIT who gave me an outstanding degree of freedom for my research during my later employment there. I would like to thank the following people for reading and commenting on drafts of this thesis. You were a great help for pointing out incomprehensible parts, spelling mistakes, and other problems that were present in earlier versions of this thesis: Goran Piskachev (Fraunhofer IEM), Eric Bodden (Fraunhofer IEM and Paderborn University).

I also thank my thesis commitee for their kind support and availability for my defense even shortly before the holidays: Johannes Buchmann, Felix Wolf, and Guido Salvaneschi. Special thanks go to the three referees for my thesis: Eric Bodden, Andreas Zeller, and Patrick Eugster. Furthermore, I would like to express special thanks to my office colleague Siegfried Rasthofer together with whom I published many papers, including many of those that built the foundation for this thesis. We always had very fruitful collaborations on various research topics and areas of shared interest in IT security. Our discussions were always academically stimulating. My special personal thanks go to Ana for the wonderful time we shared despite my workload.

The measurements with JoDroid on DroidBench would not have been possible without the kind assistance of Martin Mohr from Karlsruhe Institute of Technology (KIT). He always immediately replied to my questions and fixed several bugs in JoDroid right away. I would also like to thank everyone who has reported bugs in the FlowDroid, DroidBench or Soot open source projects, and especially those who have submitted fixes for the issues they have detected. Those external contributions that are relevant for this thesis have already been explicitly mentioned in the respective sections. Finally, I would like to thank the German Federal Ministry of Education and Research for their funding through the projects EC SPRIDE (European Center for Security and Privacy by Design) and CRISP (Center for Research in Security and Privacy).

Table of Contents

[1] This section is partially taken from our 2014 PLDI paper on FlowDroid [9].

[2] Large parts of this Section are taken (directly or with minor modifications) from our 2016 ICSE paper[5]

[3] Large parts of this Section are taken (directly or with minor modifications) from our 2016 SOAP paper[6]

1 Curriculum Vitae

The author was born in Heidelberg, Germany. He received his Bachelor's degree in computer science from Technische Universität Darmstadt, Germany in 2010. In 2012 he received one master's degree in computer science and one master's degree in IT security from the same university. During his studies, Steven spent one term (winter term 2010) at the University of British Columbia in Vancouver, Canada as a visiting graduate student.

Community Service

Steven is one of the core maintainers of the Soot open-source compiler framework that is now used for static analysis and program instrumentation by various research groups around the world. He is also actively maintaining the FlowDroid open-source static data flow tracker presented in this thesis to make it usable for other researchers and the interested public. Steven has served as the co-chair of the 2014 SOAP workshop[4]. He has been on the tool committees of the 2016 conference on Runtime Verification (RV'16), and the 2016 International Symposium on Software Testing and Analysis (ISSTA'16). Additionally, he has worked as a subreviewer for numerous conferences (FSE'13, ISSTA'15, ASE'16, CODSPY'16, FSE'16, ICSE'16, ICSE'17) and as a reviewer for several journals (IEEE Security & Privacy, IEEE Transactions on Information Forensics and Security, IEEE Transactions on Dependable and Secure Computing, ACM Computing Surveys).

Supervised Bachelor & Master Theses

- Christopher Roth: Vulnerability Analysis for Android Apps (Bachelor Thesis, ongoing, preliminary title)
- Tobias Kußmaul: Program analysis for the MS .NET framework (Master Thesis, 2016)
- Patrick Müller: Flow-insensitive information flow analysis for Android (Bachelor Thesis, 2015)
- Alexander Jandousek: A visualizer and debugger for static analyses in Eclipse (Bachelor Thesis, 2014)

Invited Talks

- Investigating State-of-the-Art Android Malware with CodeInspect (Anwendertag IT-Forensik 2016)
- Making Malory Behave Maliciously: Targeted Fuzzing of Android Execution Environments (WODA'16)
- All Your Code Belongs To Us: Dismantling Android Secrets With CodeInspect (GOTO Copenhagen'15)

Guest Lectures

- Android Security: Where does it break and how can we fix it? (University of Bergen, 2016)
- Static Analysis with Soot: An introduction into static analysis using Soot (University of Bern, 2016)
- Static Security Analysis: Challenges, Tools, and Techniques for Analyzing Android and Java (Remote Lecture for IIIT-Delhi, 2014)

Awards

- IT-Sicherheitspreis of the Horst Görtz Foundation, 1st place (2016)
- TU Darmstadt Ideas Competition ("Highest Foerderpreis"), 2nd place (2016)
- Innovators under 35, Technology Review Magazine Germany (2016)
- Special Recognition in the context of the Walter Masing Price (2007)

[4] http://www.sable.mcgill.ca/soap/2014/

2 Introduction

Today, most people use smartphones on a daily basis to perform various tasks. The smartphone has not only replaced the traditional cellphone, but also the printed calendar, the paper notebook, the camera, the alarm clock, and many other devices. For many people, the smartphone has become the indespensable single device for both private and professional use. The sales numbers of smartphones have greatly exceeded those of traditional PCs [123] and many services are now used through smartphones more often than through traditional desktop computers. Google, for instance, reports that mobile use of their search engine has surpassed desktop uses [59]. For Facebook, the number of mobile-only users has supassed the number of desktop-only users already in 2013. From 2015 on, there are also more mobile-only users than users who access the site both through a desktop computer and a mobile device [26]. These numbers indicate that this trend is likely to continue, driving more users to mobile devices. In economics, this has lead to the buzzword *mobile-first* being invented for companies that actively target their business model at mobile customers [52].

The popularity of smartphones is not only due to their built-in feature sets, but, even more important, due to their extensibility. Users can download and install programs, called *apps*, that run on their smartphone and extend its basic functionality. Some of these apps are developed by the same company that also developed the mobile phone's operating system, others are provided by the mobile network carriers or independent software companies, and yet others were developed by private people. In essence, even the basic functions of a modern smartphone such as the calendar or the notebook are, from a technical point of view, only pre-installed apps. There are apps available for practically every common need, and thousands of apps get published every day. Though all smartphone operating systems ship mobile web browsers, these browser apps are usually not the main channel through which web services such as social media, weather forecasts, or news agencies are accessed. Instead, most of these service providers offer specialized app optimized for the screen size and user interface of the smartphone. An average user of the Android operating system has 95 apps installed, 35 of which are used on a daily basis [133].

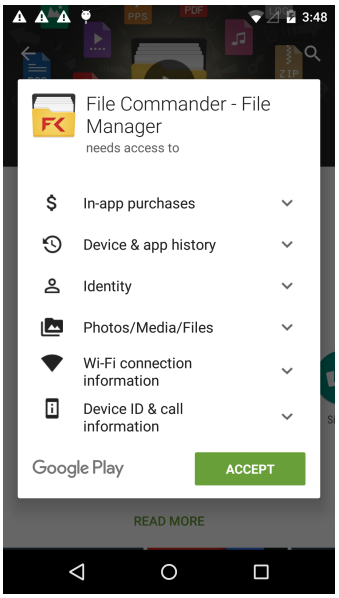
As a consequence of these trends, more and more data is processed on smartphones, including confidential and privacy-sensitive data. Users make purchases through smartphones and enter their credit card data into the device. They organize their calendars containing the names of customers they meet and projects they work on. Some also read and write business e-mails or manage their bank accounts and stocks on their phone. While it is convenient to have access to all this data and features on the go, anytime and anywhere, it also makes mobile phones interesting targets for attackers. In comparison to traditional desktop computers, smartphones are centralized data storage and processing hubs. Most apps synchronize their data with the cloud; no matter whether a new appointment is added through the calendar app or the provider's web application, it will quickly end up in the app's data storage on the phone. This centralization offers increased convenience for the user as all his data is accessible in one location, but also makes protecting this single location very important.

This is aggravated by the fact that many apps are personalized. A stock management app, for instance, not only offers to search for stock prices, but also to save certain stock the user is interested in. He can then get quick overviews over all his stock or receive alerts if prices change beyond a certain threshold. This is not only convenient for the rightful user, but also for an attacker. If an adversary manages to compromise the stock management app, he knows all of the user's stock, and potentially the banking details used to buy and sell the stock. Furthermore, smartphones not only process data entered by the user or synchronized from the cloud. These devices are also equipped with various sensors apps can make use of such GPS receivers and acceleration meters. For all this sensor data, there is a borad variety of benign use cases: For navigating on a map, the app needs to obtain the user's physical location to give correct and efficient routing information. For some games, the user has to tilt his phone to move objects which is implemented using the phone's acceleration sensor. Sensors are crucial to the modern mobile experience. On the other hand, these sensors can also be exploited by adversaries. If an attacker is able to receive regular updates on the phone's location, he obtains an accurate picture of the whereabouts and movements of its owner. This allows him to track the user through the pyhsical world. From this location data, more information can be derived: If the phone is connected to the charger at a specific location every night, this is probably the user's home location. If the phone is always in the same area during working hours, this is probably the user's office location. From these two pieces of data, an attacker can then approximate the wealth of the user using public census data. Google actively promotes this services for their advertisement business: "Target locations by demographics to reach groups of people based on their location's approximate average household income. Based on publicly available data from the US Internal Revenue Service (IRS), advertisers are able to target ads to certain areas according to their average household income." [58]

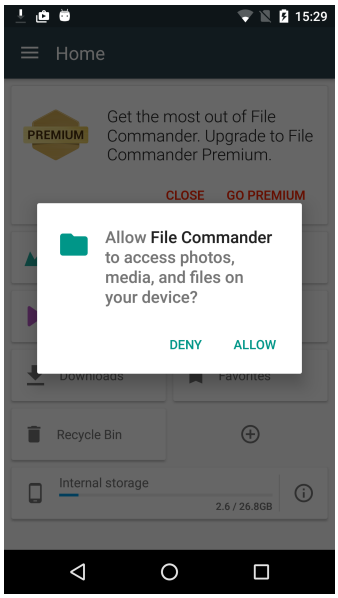
We note that for the purpose of targeted advertisement, companies not only collect data from the users while they interact with the respective company's services, but also from other sources. Facebook, for instance, collects user data whenever the user visits a website that contains the Facebook library for liking or sharing content, regardless of whether the user actually clicks on these buttons. Furthermore, many mobile apps include the Facebook SDK for displaying advertisements on their own, which gives the company even more data to build user profiles. Facebook actively promotes this data collection to advertisers: "You can even target your ad to people *based on what they do off of Facebook*." [44]. With mobile user data, the precision and completeness of such profiles for targeted advertisement are greatly improved, at the cost of user privacy and often unbeknownst to the user. With their combined data from the social network and other sources for data collection, Facebook can already now target advertisements based on location, demographics (including age and gender), interests, behaviors ("Behaviors are activities that people do *on or off Facebook* that inform on which device they're using, purchase behaviors or intents, travel preferences and more", [45]), or similarity to existing customer profiles ("lookalike audiences"). The location data is obtained from various sources, including the GPS data from apps that run the facebook app. In their official FAQ, the question "How does Facebook know when people are in the locations I'm targeting?" is answered with "Facebook uses information from multiple sources such as current city from profile, IP address, *data from mobile devices* and aggregated information about the location of friends." [44]. For Google's advertisement services, mobile data also plays a crucial role when targeting advertisements to potential customers based on demographics. On their support pages they write, "This targeting feature uses an *advertising identifier linked to a customer's mobile device* to remember *which apps the person has used*. We might associate the identifier with a demographic category *based on web browsing and app activities on a mobile device*." [57] Statistics on which apps are used how by which categories of people allow Google to more precisely associate an unknown user with a demographic category without openly asking the user for any information he might not consciously want to provide. We notice that user data, especially the data provided through always-on mobile device equipped with sensors, is a valuable currency to the advertisement companies. For many seeminly free services, the user in fact pays with his data.

Technically, the user must consent to such data collection by accepting the respective companies' data collection policies or terms of service. Usually, these terms are displayed upon first use of a service or app. However, given that these documents are long, written in legal speech, and perceived as a hurdle in front of the user's actual goal, namely using the app or service, many users simply click on the "accept" button without reviewing the policies in detail. Albeit obtaining and using as much data as legally possible without making the user too aware of the extent and implications of this business model, the renowed app developers and service providers are commonly careful to at least comply with the data protection regulations of the countries in which they operate. For smaller independent developers, this is not necessarily the case. In an earlier study [113], we found numerous cases in which such developers fail to provide even the minimum technial data protection standards. For the user, it is therefore critical to know how the apps installed on his phone actually process his data, be it from storage (calendar, address book, etc.) or from sensors such as GPS. This problem cannot be solved with regulations and policies alone, and the user cannot necessarily trust the data collection policies (if any) issued by the app developers.

In summary, smartphones offer a new way of interacting with information through a device that is not only always on, always up-to date, and personalized, but also context-aware through its sensors. Users increasingly depend on the services that are enabled by these features. On the other hand, this reliance also comes with a risk of misuse or, more generally, unwanted data leakage. Whenever a user installs an app on his phone, this app may enhance the features of his phone, but it may also contain (additional) malicious code that exploits the phone's features for an attacker's benefit. For a user, such unwanted behavior is hard to detect. Assume he installs a gaming app which actually behaves like a game, but additionally also sends out his location data in the background. For the user, there is no visible indication of this data leak. Even if the code is not outright malicious, it might serve the interests of other entities such as advertisement providers. A normal and perfectly legit app can still violate the user's reasonable expectation of privacy through advertisement libraries embedded into the app. Even the app developer might not be fully aware of the concrete data collections performed by the advertisement SDK that he embeds into his app as a binary-only library. In both cases (malware and targeted advertisement), the user is usually not aware of the data that is being collected about him, and might not consent to the data collection if he were asked openly. Additional technical measures are required to ensure user privacy and prevent data misuse, while still retaining the benefits of modern smartphones. The user must be given a means to find out how an app processes his sensitive data and to where this data (or data derived from it) is leaked. Only then, he can make an informed decision as to whether an app is trustworthy enough to be installed on his phone (and thus be entrusted with his personal data) or not.



(a) Install-Time Permission Requests in Android 5



(b) On-Demand Permission Requests in Android 6

Figure 1: Android Permission Requests Pre- and Post-Lollipop (Android 6.0)

2.1 Existing Techniques for Protecting User Privacy

The smartphone operating systems (including, but not limited to Android) already provide various techniques to mitigate the risk of data theft and misuse. Many other techniques have been proposed in academic research papers, both on the system level and on the application level. This section presents the most common and influencial approaches for protecting user privacy on Android. We show that these techniques are insufficient for truly protecting the user's privacy and argue that they mus tbe complemented with efficient and precise data flow tracking. We will show that the user must be informed not only that his data is bing used, but where it is passed and leaked, which goes beyond traditional access control techniques.

2.1.1 Permissions

The Android operating system enforces an explicit permission model. Sensitive operations such as reading out the current GPS position are guarded by permission checks. An app must declare a list of all permissions that it will ever use during its execution. In the traditional Android permission model (up to version 5), this list is presented to the user when he installs the app as shown in Figure 1(a). If the user does not want to grant the app any one of these permissions, he can abort the app installation and completely refrain from using the app. There is no possibility to deny individual permissions and still install the app. Apps are guaranteed to never use any permissions they haven't declared. Therefore, an app is always limited to the permissions granted by the user at install time. Nevertheless, this model does not prevent data leakage for multiple reasons. Firstly, many users do not carefully inspect the permission list, but instead simply accept it as their only other chance would be to refrain from using the app at all [48]. Even those users who read and inspect the permission list often fail to correctly map the requested app permissions to potential security and privacy threats [73]. Secondly, permissions are coarse-grained. If an app requests the permission to connect to the Internet, this does not restrict the remote servers to which connections are made and to which data is potentially transferred. Approaches for fine-grained permissions have been proposed in the literature [69], though. Lastly, permissions only control access, but not usage. If an app is granted access to the phone's GPS location, there is no further restriction on what the app can do with this data. A benign app could display the position to the user on a map whereas a malicious app could silently leak the data to a remote adversary. The user cannot distinguish these cases once he grants the permission to access the GPS location. Therefore, allowing the user to remove permissions as proposed by Do et. al [39] or as implemented in AppGuard [16] is not a solution either.

In the most recent Android versions (starting from 6.0), the permission model was switched to an on-demand permission check [3]. Users are no longer requested to accept or deny all permissions as an all-or-nothing operation at install time. Instead, whenever an app tries to perform an operation guarded by a permission, a corresponding request is presented to the user as shown in Figure 1(b). This gives the user more contextual information, i.e., he can decide whether this very action matches his expected app behavior at this time. If a timetable app for public transport, for instance, suddenly requests his GPS position, this is supicious and should likely be denied - unless there is a good reason such as a *find nearest bus stop* feature on which the user just clicked. Even on-demand permissions are not a complete solution to the problem of data leakage, though. Firstly, the decision as to whether a certain permission request is expected or not, can be hard to make. If the timetable app, for instance, displays the nearest bus stop right away without any prior user interaction, it might not be clear why it asks for the GPS location directly on startup. Furthermore, even on-demand permission requests do not restrict the use of the data once it has been obtained by the app. This is especially important for apps that rightfully request sensitive data such as a GPS navigation app. For the user, there is a good reason to always grant this app access to his location while using it. In addition to the benign route planning feature, the app is, however, still free to maliciously leak the data to an adversary without being noticed by the user.

Researchers have evaluated whether combinations of permissions are effective to mitigate privacy issues [40]. If an app wants to leak the GPS data, it conceptually needs two permissions: One to access the GPS data and one to send it out to the Internet. One could, therefore, disallow apps that have such dangerous combinations of permissions. In practice, however, these combinations can also occur in benign applications such as the GPS navigation app mentioned above: It needs the user's location for routing and Internet access for downloading maps. Furthermore, even the absence of the Internet permission is no guarantee that no data is leaked to the Internet as researchers have shown [134]. In total, the permission system alone is insufficient to ensure user privacy and prevent data leakage from smartphones. Instead of pure access control on sensitive information, data flow control is needed to check for misuses of legally obtained data.

2.1.2 Context-Sensitive Access Control

The Pegasus system [30] associates events with permission-protected API calls to form a *Permission Event Graph*. The idea is that a user encodes his expectations of when certain APIs are used as policies. The Pegasus tool then uses a combination of static and dynamic checks to detect violations of that policy. While this approach can detect malicious behavior or unexpected private data leaks in the presence of precise specifications, it is not readily applicable to arbitrary apps. The authors present simple application-independent policies such as "Sensitive operations must be guarded by user interaction". An arbitrary user interaction does, however, not automatically induce consent on an arbitrary follow-up operation conducted by the app. Approaches such as Pegasus therefore fail to inform the user about the concrete sensitive operation (e.g., data leak) to which he is consenting. Creating fine-grained policies that capture such semantics, on the hand, is a time-consuming and non-trivial task that cannot be expected from an end-user. Together, these issues limit the practicality of the approach in real-world scenarios.

2.1.3 Mock Data

Some researchers have proposed to transparently replace the actual sensitive data items stored on the phone with fake mock data [20]. A malicious app would then still be able to leak this data, but since all data is randomly generated, it is useless to an attacker and does not infringe upon the user's privacy. This technique works under the assumption that such data accesses are not central to the app's core functionality. If a GPS navigation app also leaks the user's location data in the background, a mocking approach cannot help. Replacing the GPS location with fake data would also lead to a wrong map display and unhelpful routing instructions and therefore destroy the app's core features. Other approaches such as AppFence [65] combine mock data with traffic blocking. Certain data items can be declared as mocked, others as app-local. The latter are available to the app in their original form, but attempts to send this data to remote entities are blocked. This, however, already requires knowlege of the data flows inside an app to notice the transmissions to be blocked. Again, data flow tracking appears to be central to protecting user privacy on smartphones.

2.1.4 Store-Based Policy Enforcement

Another approach to mitigating privacy risks on smartphones is to control the app distribution. If malicious apps are not available for installation, they cannot cause harm to users. Such control is possible, because the distribution

of apps is largely centralized which is a major difference to traditional computer software which can be obtained through various channels (CDs/DVDs, download from various websites, etc.) All big developers of smartphone operating systems have created ecosystems around their systems and the apps offered for them. At the center of such an ecosystem, there is a *store*. For the Android operating system, Google offers the *Google Play Store*, for iOS by Apple, there is the *Apple App Store*. These stores the main distribution points of apps. Developers can upload their new apps to the store and manage their updates and patches through it. Users can search the stores to find a suitable app for their particular needs, usually through the store app pre-installed on all of the respective manufacturer's phones. This store app then also manages the installation of the app and notifies the user of available updates.

App store providers can filter which apps they allow into their stores and ban those apps that have been identified as malicious. This, however, imposes additional effort on the store operator. Worse, a binary decision of whether to allow or reject a newly uploaded app also requires a shared understanding of wanted and unwanted apps between store operator and smartphone user. While a store operator might accept apps that create and leak user profiles for advertisement purposes as part of his own revenue model, such data leaks may nevertheless be considered unwanted by the smartphone user. Furthermore, especially the Android ecosystem is not restricted to the official Google Play Store. Users can optionally chose to also install apps from other sources. There are various third-party app stores run by network carriers or independent third-party organizations. In some countries such as China, the Google Play Store, for instance, is not even available, forcing users to obtain their apps through other stores. Each store has different policies on what kinds of app behavior are acceptable and not even all third-party app stores perform malware checks at all. Therefore, traditional binary assessments at the store level (i.e., allow an app into the store or reject it) are not a full remedy against privacy violations either.

Instead, new techniques for detecting and enumerating flows of potentially sensitive user data are required. The user not only needs to be informed which kinds of data are accessed by an app, but also where this data is transferred to. If an app reads out his GPS location, what happens with this data after it leaves the protection of the smartphone operating system such as Android's permission model? Judging the behavior of the app based on this information is and needs to be the task of the well-informed user as only he knows his personal privacy requirements.

2.1.5 Data Flow Analysis[6]

Other researchers have also identified the need for data flow analysis on mobile apps and created various analyzers, both before and after the work described in this thesis was made available as open-source projects. In this section, we will discuss those works that are closest to our own work in the area. Section 5.5 will give a broader overview over the area of static data flow analysis in general and will also discuss related work on some of the important building blocks necessary for analyzing Android apps.

Note that confidentiality and integrity are dual problems from for a data flow analysis. Both are essentially flows between API calls. In one case, confidential data flows to untrusted sinks, in the other case untrusted data flows to sensitive sinks. In this area of integrity, CHEX [91] is an approach for finding component hijacking vulnerabilities in Android. Although not built for the task, CHEX can, in principle, be used for taint analysis. CHEX does not analyze calls into Android framework itself but instead requires a (hopefully complete) model of the framework. In our work, such a model is optional and mainly used to increase performance. We also provide for techniques to create library models which is missing from CHEX. Furthermore, CHEX is limited to at most 1-object-sensitivity, while the demand-driven alias analysis of the work presented in this thesis allows for contexts of arbitrary lengths (using a default of 5). We found 1-object-sensitivity to be too imprecise in practice.

LeakMiner [153] appears similar to the work presented in this thesis from a technical point of view. It is based on Soot, uses SPARK for callgraph generation, implements the Android lifecycle, and the paper states that an app can be analyzed in 2.5 minutes on average. However, the analysis is not context-sensitive which precludes the precise analysis of many apps as we will point out in later sections of this thesis. AndroidLeaks [55] also states the ability to handle the Android Lifecycle including callback methods. It is based on WALA's [141] context-sensitive System Dependence Graph with a context-insensitive overlay for heap tracking, but is not as precise as our work, because it taints the whole object if tainted data is stored in one of its fields, i.e., is neither field nor object sensitive. This precludes the precise analysis of many practical scenarios. SCanDroid [54] is another tool for reasoning about data flows in Android applications. Its main focus is the inter-component (e.g. between two activities in the same app) and inter-app data flow. This poses the challenge of connecting intent senders to their respective receivers in other applications. SCanDroid prunes all call edges to Android OS methods and conservatively assumes the base object,

[6] This section is partially taken from our 2014 PLDI paper on FlowDroid [9].

the parameters, and the return value to inherit taints from arguments. This is much less precise than our work which can use automatically-generated precise library summaries for modeling calls to framework methods, or, alternatively, analyze the framework code together with the app. Our approach currently models intent sending as sink and intent reception as source, yielding a sound treatment of inter-app communication. We also desribe how our work has already been extended to provide for true inter-component and inter-app modeling and data flow analysis.

More recent approaches such as DroidSafe [60] tackle many of the precision and recall issues of prior work. Especially DroidSafe, however, requires substantial human labor as it needs stub implementations of the complete Android framework, including all the Java base classes. Furthermore, we found DroidSafe to suffer from serious scalability issues even on small micro-benchmarking apps. We were therefore not able to analyze all apps we initially inteded with DroidSafe. JoDroid [97] is an extension that adds support for Android apps to the Joana [56, 61] object-sensitive analyzer for Java. Like the original Joana work, it is based on program dependence graphs (PDGs) [50]. This program representation allows for very precise reasoning on data flows including implicit flows, but can also be costly to construct. Our approach in this thesis is based on taint tracking based on an inter-procedural control flow graph. ScanDal [75] is a static data flow analyzer based on abstract interpretation. Unfortunately, the implementation tool is not publicly available for a direct comparison.

Other approaches like CopperDroid [115] dynamically observe interactions between the Android components and the underlying Linux system to reconstruct higher-level behavior. Special stimulation techniques are used for exercising the application to find malicious activities. Attackers, however, can easily modify an app to detect whether it is running inside a virtual machine and then leak no data during that time [93, 106, 139]. Alternatively, data leaks might only occur after a certain runtime threshold or when a specific sequence of events has been triggered. The limited code coverage is a well-known problem of dynamic approaches that require event simulation. A recent comparative study of test input and event generation tools for Android [31] shows that even those tools that achieve the highest coverage can only yield a statement coverage of less than 50% on average. In other words, more than half of the statements are never triggered when automatically exercising an Android app and all behavior inside that part of the app remains hidden to the dynamic analysis tool. Aurasium [147] and DroidScope [149] largely suffer from the same shortcomings in comparison to static leak detection.

TaintDroid [41] is one of the most sophisticated taint tracking systems for Android to date. As a dynamic approach, however, it yields some quite different tradeoffs compared to the work presented in this thesis. For instance, TaintDroid has no problem tracking taints through reflective method calls, as TaintDroid is implemented as an extension to the execution environment, for which it does not matter whether methods are invoked through reflection or not. On the other hand, if used for triaging malware before installation time, then TaintDroid can successfully detect malware only if paired with a dynamic testing approach that yields decent code coverage. Obtaining such coverage is, as mentioned above, a research challenge on its own, for which no satisfactory solutions exist yet. Static ahead-of-time analyses like our work do not share this shortcoming because they, by their nature, cover all execution paths. Secondly, a dynamic approach such as TaintDroid can be fooled by a malicious apps that recognize that it is being analyzed in which case the app could simply refrain from performing any malicious activities [93, 106, 139]. While this is not problematic if the dynamic analysis is installed on the end user's mobile phone (in that case, the malware would effectively be tamed), it is problematic if the dynamic analysis is only used for ahead-of-time triaging of malware that could then later on be installed on a system not protected by the dynamic analysis (in which case the app could resume its malicious activities). Static approaches such as ours do not share this particular shortcoming as they never actually execute the app.

F4F [130] is a framework for performing taint analysis of framework-based applications using a specification language called WAFL for describing the functional behavior of the respective framework. While originally created for web applications, it might also be extended to model the Android framework by adding a WAFL generator for Android. The dummy-main generation technique we propose in this thesis has the big advantage to only include components and callbacks that are indeed accessed by the app. This, however, requires a semantic model of the app's manifest, the layout XML files, the compiled resources file and the appâĂŹs source code, which are all interleaved. F4F could at best be used to give a coarse approximation modeling the common denominator of all possible apps.

While many of the approaches presented above are important steps towards the goal of informing the end-user about the data flows inside an Android app, none of them solves the challenge completely. For a data flow analysis to be usable in practice, it must be efficient and simple to use. The dynamic analyses such as TaintDroid, Aurasium, and DroidScope require the user to fully exercise the app in the analyzer, which is infeasible. We therefore propose static analysis to fully capture the behavior of an app while avoiding the code coverage problem. The existing static

analysis tools, on the other hand, suffer from low precision, which requires the user to review a large number of flows in the end, many of which are false positives and do not model actual behavior of the app. Approaches based on PDGs have the potential to be more precise, but suffer from high computational cost. In fact, as we show in Section 7.2.5, some of the tools time out on relatively small benchmarks app with less than one megabyte of total size and only a handful of lines of actual user code. Other tools require significant human effort, which is infeasible if one wants to analyze a complete app store. In the case of a user who is not an expert on (static) analysis on his own, such a requirement may also be impossible to meet. Yet other tools claim interesting properties, but are not publicly available for indepedent validation of their claims and a fair comparison with new ideas and approaches. In summary, there is still the need for a static data flow tracker that efficiently, precisely, and as completely as possible, analyzes real-world Android apps. In this thesis, we propose FlowDroid exactly as such a tool, which we make available as an open-source project for other researchers to try out, extend, build upon, and independently validate our claims. We purposely design the tool with many interfaces for easily replacing algorithms and techniques with new approaches in an attempt to provide the community with a testbed for individual building blocks.

2.2 Thesis Statement

In this thesis, we explore whether static data flow analysis can be effectively and efficiently used to detect privacy leaks in Android apps. We aim to construct a tool (or a suite of tools) that covers all necessary steps in the analysis process such that a user can input an app and is, as a result, presented with a list of data flows of potentially sensitive information inside the given app. To be useful in practice, the tool must be scalable and fast enough such that it can be applied to popular real-world apps from major app stores, which also happen to have large code sizes for the examples we have observed. Secondly, the analysis must discover a high percentage of the leaks that exist inside the app to be analyzed, because each missed leak is a potential privacy violation the user is not aware of. Lastly, the tool must be as precise as possible to not overwhelm the user with false reports on data flows or privacy leaks that actually cannot occur in practice. In the following sections of this thesis, we show that such an analysis is indeed possible and present a design and implementation of a highly precise and efficient static data flow tracker as well as techniques for capturing Android-specific semantics and handling large libraries. Our final result has gained significant popularity in the research community and has been used as a component in many derived works.

Note that we do not judge the discovered data flows in this thesis, i.e., do not decide whether a given data flow violates the user's privacy or is his reasonable assumptions about what data an app will share and what it will keep private. We do not reason about flows being benign or malicious either. We acklowedge that such decisions are important and need to be made, but leave this as a separate research question. Still, because such judgement is an important next step in order to increase the helpfulness of the work presented in this paper for real-world end users, we shortly present a few approaches developed by other researchers. The same holds for defining the sources of sensitive data in the Android environment and the sinks that may potentially leak this data to untrusted outsiders. In this thesis, we survey techniques for obtaining this required configuration, but do not present own approaches.

Android has long since become the most popular smartphone operating system with a market share of about 86% [131]. Furthermore, the Android operating system is available as an open-source project, which makes Android readily available for research and scholarly inspection. Other operating systems such as iOS and Windows Phone instead require a great deal of reverse engineering effort for understanding their inner workings. For these reasons, the system-specific parts of this thesis are centered around Android. Note that the general concepts can be applied to Java programs completely unrelated to the mobile domain which we will also highlight in the later sections of this thesis.

2.3 Contributions

This thesis is a step toward better protecting the users' privacy by detecting potentially undesired data flows in apps. We present techniques aiming towards this goal as well as infrastructure components that make the analysis itself feasible. In summary, this thesis presents the following original contributions:

- We design and implement a static data flow tracking tool that is both efficient and highly precise. This tool, called FlowDroid [9] consists of a generic, platform-independent data flow tracker and platform-specific

extensions. This design not only allows to the tool to be applied to multiple platforms such as Java and Android, but also allows for its integration into other research projects. The tool is presented in Section 4[7].

- We design and implement extensions to FlowDroid that faithfully model the properties of the Android operating system. Our model includes the Android lifecycle, callbacks, and UI elements. As Android is a highly dynamic platform, we carefully chose trade-offs for the static approximations of Android runtime behavior. The Android extensions to FlowDroid are presented in Section 5.

- We design and implement a technique called StubDroid [5] for automatically generating method summaries from a binary distribution of a library. We further implement techniques for making these libraries available to FlowDroid. As a result, FlowDroid can skip analyzing these libraries every time anew when processing an app. Instead, the pre-computed summaries are plugged in. We demonstrate that the use StubDroid summaries for the Java collections API on Android and Java reduces the runtime of the data flow analysis by over 90%. StubDroid is presented in Section 6.

- We design a test suite called DroidBench for assessing the precision and recall of static and dynamic data flow analysis tools. This suite helps in comparing various approaches from academia against each other and against commercial products in the field using reproducible experiments on publicly available data. DroidBench consists of over 100 small Android apps with challenges such as aliasing or data propagation over the Android lifecycle. DroidBench is presented in Section 7.

- We show that, while FlowDroid is neither totally sound nor totally precise, the tool delivers useful results in practice with a precision of more than 86% and a recall of more than 79% on realistic micro-benchmarks from DroidBench. Even on large real-world apps such as Facebook , the analysis finishes in about 2 minutes, while taking even under one minute for many other apps.

- We present a first step toward broadening the scope of the FlowDroid static data flow analyzer and the Soot compiler framework on which it is based beyond Java and Android. In this work, presented in Section 9, we show how assemblies compiled for the Microsoft .net framework can be converted to Jimple code. This is a necessary prerequisite for applying our existing tool chain on this platform as well which we plan as future work.

2.4 Dissertation Outline

The goal of this dissertation is to show whether and if so, how, static data flow analysis can be used to analyze Java programs and Android apps for detecting leaks of privacy-sensitive data. The remainder of this thesis is structured as follows. In Section 3, we explain the general concepts on which our work is based. This includes a discussion of the frameworks used as well as general problems such as how to define sources and sinks that capture the user's intuition of what sensitive data is and where it should not be leaked. We explain how access paths, the primary abstraction for representing taint inside the static data flow analysis, are structured and motivate why we chose this representation. Afterwards, we give a short introduction into the IFDS framework in which FlowDroid's data flow problem is formulated. We also discuss the fundamental limitations of the chosen frameworks and representations.

Section 4 presents the platform-independent core of the FlowDroid data flow tracker, i.e., the parts that are not specific to Android, but that can be applied to Java (and potentially many other platforms) as well. We discuss how precision, scalability, and flexibility can be achieved, and what tradeoffs were made. We describe each of the components of FlowDroid in detail including its features and limitations. We also give an outlook on some of the work other researchers have done based on FlowDroid. Afterwards, Section 5 presents the Android-specific extensions to this core data flow tracker. We describe how we handle the Android lifecycle, the callback-driven nature of Android app programming and execution, and the tight coupling that many apps have with their user interface. Together, Sections 4 and 5 reflect the full FlowDroid system as it is available to the research community as an open-source product.

In Section 6, we discuss StubDroid, a technique for handling large libraries in a static data flow analysis. We show how one can create efficient summaries for the Java base libraries and the Android SDK to avoid having to re-analyze these common classes together with every app or program. We show that our library-based approach

[7] An initial prototype of FlowDroid was created by Christian Fritz in his master's thesis [53]. The version presented in this Ph.D. thesis improves significantly over the initial prototype in functionality, efficiency, precision, and recall.

can reduce the overall analysis time by up to 80%. For evaluating static data flow trackers such as FlowDroid, we provide a suite of micro-benchmarks called DroidBench in Section 7. With this suite that has gained enough popularity to be used for evaluation in various research papers, we show how FlowDroid performs in comparison to other approaches available commercially or known to the scientific literature. Since these micro-benchmarks do not give insights into a tool's scalability for real-world apps, we evaluate FlowDroid on a number of popular apps taken from the official Goole Play Store in Section 8. We first report on time and memory consumption with the default settings, and then evaluate how the most important settings affect the scalability of the tool on our experimental set of apps.

In Section 9, we present an outlook on work to extend FlowDroid and the Soot compiler framework on which it based beyond Java programs and Android apps. We present a framework that transforms code compiled for the Microsoft .net framework in to Soot's Jimple intermediate representation. This serves as a first important step towards applying FlowDroid to this platform as well in future work. We finally conclude our discussion of static data flow analysis in Section 10.

3 Concepts of Static Data Flow Analysis

In this section, we will explain several important concepts that are necessary for the techniques and methods we propose in the remainder of this thesis. For reducing the complexity of the analysis, we do not conduct our static analyses directly on the Java or, in the case of Android, Dalvik bytecode, but rather use a simplified intermediate-representation called Jimple which we explain in Section 3.1. Section 3.2 focuses on defining sources and sinks as an integral part of the data flow problem. In Section 3.3 we discuss access paths, which are the fundamental concept of our taint abstractions. The FlowDroid static data flow tracker is implemented as an IFDS problem. Therefore, we give an introduction into the IFDS framework in Section 3.4.

3.1 The Jimple Intermediate Representation

Analyzing Java or Dalvik bytecode is complex, because these languages have been optimized for runtime execution performance, not for static analyis. Java bytecode has more than 200 different opcodes that an analysis would all have to emulate with respect to the data flow facts. For each opcode, the analysis would need to have rules on how to transform an incoming data flow fact such that the output resembles how the opcode has influenced the respective data. For such a large numbner of opcodes, this would be a major undertaking. To simplify the analysis, the work presented in this thesis is therefore based on the Soot compiler framework [78] and its Jimple intermediate representation [137]. Jimple consists of only 15 different types of statements that can contain 29 different types of expressions. While for performance reasons Java bytecode, for instance, has individual opcodes for putting one of the integer values `0, 1, 2, 3, 4, 5` on the stack, there is no such special-casing in Jimple. Furthermore, Java bytecode is a stack-based language. Operands are put on the stack before executing an operation. The operations pops the arguments, computes the results, and puts these results back on the stack. A static analysis working on this level would have to emulate this complete stack semantics. In Jimple, there is no stack. Instead, Jimple is closer to the Java sourcecode language, because it uses `locals` which are simply local variables that can be read from and assigned to. In total, Jimple can be seen as a higher level of abstraction than bytecode, though there is a complete mapping from bytecode to Jimple and back again. Unlike with decompilation to Java source code, there is no semantic loss when creating Jimple code from Java bytecode.

Jimple inherits some properties that are relevant for static analysis directly from the bytecode language. While the Java source code language supports complex nested constructs, both the bytecode and the Jimple language break them into a sequence of simpler statements. Adding more than two numeric values, for instance, is broken into a sequence of additions with two values each. As another example, if the original source code invoked a method and passed a value from a field as a parameter, i.e., had a nested field access inside a method call, this is broken into two different statements as well. First, the field is read and the field value is written into a temporary variable. Then, the method call takes place with the value from the temporary variable. While these simplifications increase the overall code size, they create code with a simple structure, effectively reducing the number of cases that must be handled in the static analysis.

3.2 Source and Sink Definition

The goal of a data flow analysis is to find connections between sources and sinks. This requires a definition of what constitutes a source and what constitutes a sink. For the use case of detecting privacy leaks, the user or analyst is, informally speaking, interested in whether privacy-sensitive information is leaked to potentially untrusted external parties. This is, for example, the case if an app reads the user's address book from his phone and transmits it to a server on the web. In this example, the source is the method that reads the address book data and the sink is the method that transmits this data (or any data derived from it) to the remote server. In the simpler example from Listing 1, the user's device ID is read and send out as the text of an SMS message. There, the `getDeviceId()` method (called on Line 4) is the source and the `sendTextMessage` method (called on Line 8) is the sink. Before any data flow analysis can be conducted, these source and sink methods must be identified. They constitute which flows the analyst is interested in (as opposed to, e.g., the flow of a constant value into a message displayed to the user which is usually not of any interest).

On Android, the only possibility for an app to access data from the operating system (such as sensor data like GPS location or device identifiers) or from other apps (such as the address book) is through API calls. The goal of identifying sources and sinks can thus be reduced to the task of partitioning the set of publicly accessible Android API methods into ones giving access to sensitive information (i.e., sources), ones allowing to send out information

```
1 void onCreate() {
2     // Get the data
3     TelephonyManager mgr = (TelephonyManager) this.getSystemService(TELEPHONY_SERVICE);
4     String deviceId = mgr.getDeviceId();
5
6     // Leak the data
7     SmsManager sms = SmsManager.getDefault();
8     sms.sendTextMessage("+49 1234", null, deviceId, null, null);
9 }
```

Listing 1: Simple Data Leakage Example. Adapted from the *DirectLeak1* test case in DroidBench

(i.e., sink), and remaining set of non-interesting methods. The easiest attempt would be to conduct this partitioning by hand, which is, however, practically infeasible. Version 4.4 of the Android operating system contains more than 110,000 public API methods, newer versions even more. Manually checking whether each of them returns privacy-sensitive information or can potentially be used to send such information to third parties is clearly infeasible. Therefore, approaches to automatically identify sources and sinks have been proposed. SuSi [109], for instance, uses machine learning to take a small set of hand-annotated source and sink methods as a base for identifying other, similar methods. Source methods share the characteristic of reading data from a system resource and returning them to the caller, regardless of the concrete type of data. Such a system resource could, for instance, be a database stored on the phone. Sink methods, on the other hand, write to system resources such as the network. SuSi is highly effective in exploiting these common characteristics for identifying new, previously unknown sources and sinks. For Android 4.4, it achieves a precision and recall of more than 92%. Many of the source and sink methods identified by SuSi were previously not included in the source and sink lists of static analysis tools, neither from academic research, nor from industry. It is important to note that every missed source or sink method is a potential security threat as attackers can call those methods to obtain and send out privacy-sensitive data items without the leak being detected. Data flow analysis tools can only be effective if their source and sink lists are complete. This again underlines that automation is necessary to properly safeguard user privacy.

Automatic detection of sources and sinks through machine learning has the potential to identify a large number of sensitive methods. As all approaches based on machine learning, it is, however, inherently incomplete. One cannot give any guarantees on the absence of false negatives. In other words, there may always be a sensitive method that is not classified as a source or sink, although it actually is one. The authors of the DroidSafe project [60], for instance, manually investigated the methods called by the apps in their test set and found that SuSi missed 53% of the source methods and 32% of the sink methods called by these apps. They therefore opted to completely manually assemble the list of sources and sinks for their analysis. As this requires a manual inspection of every app before running the actual data flow analysis, it can, however, only be seen as a supplementary approach for a fully-automated large-scale analyses and not as a replacement.

While SuSi exploits implementation properties of sources and sinks as features for its machine learning algorithm, other approaches exploit similarities in how these methods are used in client programs. Merlin [89] starts from an initial specification much like SuSi's hand-annotated training set. It then looks for similar data flows in a set of client programs to find further, not yet identified, sources and sinks. According to the original paper, Merlin's false positive rate is 6% for sources and 26% for sinks. This approach, however, can only identify those sources and sinks that are used in at least one of the applications in the analysis set. While this will, given enough apps, usually cover all the common sources and sinks, it still misses the more "obscure" (i.e., rarely-used) methods. This leaves more room for attackers to deliberately exploit those rare methods for conducting undetected data leaks.

AndroidLeaks [55] is representative for yet another approach to identifying sources and sinks, namely through Android permissions. The idea is that most methods that give an app access to sensitive information are protected by permissions. If an application, for instance, wants to read out the user's location data, it needs the `ACCESS_FINE_LOCATION` or `ACCESS_COARSE_LOCATION` permission. Given a mapping between API methods and the set of permissions required by these methods, one only needs to collect all methods that require one of the well-known information-retrieval permissions. Several research projects have focused on providing such permission-to-method mappings [13, 47]. While this approach is, given such a mapping, trivial to implement for sources, it is more complicated for sinks. AndroidLeaks cannot easily rely on permissions here, because the `INTERNET` permission is enforced by the kernel-level sandbox and not by the Dalvik-level Android framework for performance reasons. For sinks, AndroidLeaks therefore resorts to a manually- compiled sink list.

```
1 void onCreate() {
2     // Get the data
3     Data d = new Data();
4     d.a = 5;
5     String x = source(d);
6     sink(d.a);
7 }
```

Listing 2: Source method call with complex objects

All approaches mentioned so far consider full methods as sources or sinks. While this definition is conceptually simple, it is imprecise. Consider the example in Listing 2 where the method `source()` is considered as a source method. Without any further information, an analysis does not know whether this method returns the sensitive data into the `x` variable or whether it writes that data into a field of the `d` object which gets passed in as a parameter. If the analysis needs to be complete, it has to conservatively over-approximate and assume that both the return value and all of the fields inside the data object receive sensitive data. While this assumption will ensure that the analysis never misses any sensitive data obtained through the `source` method, it can also lead to serious *over-tainting*. In other words, it can make the analysis track spurious data elements that are not sensitive at all. In the example, constant value stored in `d.a` might not be overwritten by the call to `source`; falsely assuming that the `source` method can taint everything in its scope will eventually lead to a false positive. When taking static fields into consideration, conservative over-approximation fails entirely as one would need to assume that every call to a source method writes sensitive data into every static field in the whole program, because static fields are always in scope. In other words, once a source method was called, all static fields would be assumed to contain sensitive information. While such a behavior of a source method is theoretically possible, most source methods will not behave that way. In most cases, an analysis that makes such a conservative assumption will suffer from a large number of false positives.

To completely solve this issue, one needs to extend the definition of a source beyond mere method signatures. Instead, the specification of a source or sink must precisely pinpoint the variables (and potentially the concrete fields reachable through these variables). For the `source` method, this could be "`taints the return value`", or "`taints <parameter0>.a`" instead of just "`method source is a source`". One example of a specification language for sources and sinks that allows such fine-grained definitions is RIFL [42]. Note that a source or sink definition has to be in relation to the externally-visible interface of the respective method method. An analysis tool that only works on client programs and that does not have access to the implementation of the source or sink method inside the framework or OS code must be able to map the source or sink definition to a concrete taint to be propagated onwards.

If the source and sink definitions are not written by hand, such fine-grained information is, however, usually not available. SuSi's classifier cannot reason about field accesses or access paths. Mapping Android permissions to methods also only yields a set of methods, but not any finer-grained information. A practically-usable static analysis tool must therefore also provide for useful heuristics in the case of coarse-grained specifications. Conservative over-approximation is usually not acceptable due to the issue of over-tainting explained above. Therefore, one usually assumes that a source method only taints its return value which is true for all of the commonly-used Android API methods, but may fail in other contexts or for future additions to the API.

Orthogonal to identifying sources and sinks is the problem of classifying them. Data flow analyses are often used inside semantic frameworks that check or enforce data-driven policies. In the original example 1, the user wants to know that his *IMEI number* is transferred to the *internet* as opposed to, e.g., his *files* being stolen via a *Bluetooth connection* to another device. This requires sources and sinks to be associated with categories such as *unique identifier*, *address book* or *files on devices* for sources. For sinks, potential categories can be *Internet*, or *Bluetooth*. When manually identifying sources and sinks, categorization is straightforward. However, the SuSi approach also supports categorization by once again applying machine-learning (just with a different set of features and new training data) to the previously identified sources and sinks.

3.3 Access Paths

Java programs and Android apps store data in and read data from fields. A static data flow analysis tool must model such heap accesses in its taint abstraction. In the example in Listing 3, the field `fld` of base object `a` gets tainted. Only the statement in line 5 accesses this heap object and therefore constitutes a real leak. The other calls

```
1 void onCreate() {
2     A a = new A();
3     A b = new A();
4     a.fld = source();
5     sink(a.fld);  // true leak
6     sink(a.foo);  // no leak
7     sink(b.fld);  // no leak
8 }
```

Listing 3: Heap Analysis for Java (1)

```
1 void onCreate() {
2     A a = new A();
3     a.fld = new B();
4     a.fld.data = source();
5     sink(a.fld);  // debatable
6     sink(a.fld.data); // true leak
7     sink(b.fld.foo);  // no leak
8 }
```

Listing 4: Heap Analysis for Java (2)

to the `sink` method should not be detected as endpoints of data flows as this would indicate false positives. In the program analysis literature, several techniques for representing taints on heap objects have been proposed:

- **Field-Insensitive Analysis:** When a field gets assigned a tainted value, the field-insensitive analysis taints the complete base object. In the example, such an analysis would represent `a.fld` by just `a`, which would not only taint `a.fld`, but also unrelated fields of the same base object such a `a.foo`. It would therefore lead to a false positive in line 6.
- **Field-Based Analysis:** When processing an assignment to a field, the field-based analysis technique taints the declared field irrespective of the base object. It does not distinguish between different objects of the same class holding this field. In the example, it would represent `a.fld` by just `A.fld`. Consequently, it would therefore lead to a false positive in line 7.
- **Field-Sensitive Analysis:** If tainted data is assigned to a field, the field-sensitive analysis taints the combination of base object and field. In the example, it represents `a.fld` by `a.fld`. When the analysis encounters a field access, it matches both elements and thus avoids both types of false positives in the example.

For the example in Listing 3, field-sensitive tracking is the most precise option. In general, even field-sensitive tracking can, however, lead to false positives. Consider the extended example in Listing 4. In this example, the sensitive data is not directly assigned to a field inside the base object `a`, but instead to a field `data` that is in turn stored in a field `fld` of the base object `a`. In other words, this example has one more level of indirection. For accessing the sensitive data from base object `a`, not only one, but actually two consecutive field dereferences are required. The call to `sink` in line 6 leaks the correct data, whereas the call in line 7 does not constitute a leak as no sensitive data is actually passed into the sink. Whether the call in line 5 shall be considered as a leak or not depends on the semantics of the `sink` function, i.e., whether it is expected to read the `data` field containing the sensitive data or only some other unrelated field. This again refers to the discussion of precise source and sink definitions in Section 3.2. The traditional field-sensitive analysis, however, can only model one base object and one field dereference, effectively tainting `a.fld` in line 4. For not missing the true leak in line 5, the analysis must consider everything that is read from `a.fld` as tainted. The analysis cannot distinguish between `a.fld.data` and `a.fld.foo`, therefore causing a false positive in line 7. Regardless of the precise sink definition, it would also always have to consider the call in line 5 a leak.

To retain precision in the case of multiple dereferences, one therefore needs an abstraction that can capture a base object plus a (possibly empty) sequence of fields. This data structure is called an *access path* [35, 135]. In the example, the analysis would taint `a.fld.data`, thereby avoiding the false positive in line 7 as well as offering maximum precision for the sink-dependent case in line 5. Note that access paths can also model taints on static fields. In this case, the base object is null and the first field in the access path's sequence of fields is assumed to be static.

In theory, an access path can precisely pinpoint a heap object by exactly denoting a series of field dereferences that provide access to the object. It is the task of the alias analysis to take one access paths and then enumerate all other access paths that can possibly point to the same runtime object. Extending the definition of an alias to access paths is straightforward. Computing aliases on access paths is not trivial, though. We will discuss this problem in detail in the context of the FlowDroid static data flow tracker in Section 4.8. Regardless of the concrete alias analysis, the set of possible access paths pointing to the same heap object can, however, be infinite. Consider the example in Listing 5. Lines 6 to 8 are accesses to three of the infinitely many aliases of `a.data`. The key

```
1  void onCreate() {
2      A a = new A();
3      a.next = new A();
4      a.next.prev = a;
5      a.data = source();
6      sink(a.data); // leak
7      sink(a.next.prev.data); //leak
8      sink(a.next.prev.next.prev.data); // leak
9      sink(a.next.prev.foo);  // no leak
10 }
```

Listing 5: Recursive Access Paths

observation is that the field dereferences are circular: `a.next.prev` is the same as `a`. Therefore, one can derive an arbitrary number of aliasing access paths by adding more walks through this loop, growing the individual access path infinitely. Traditionally, this issue has been solved by *k-limiting* [71]: The maximum length of all access paths inside an analysis is set to a constant value `k`. Whenever the analysis generates an access path that exceeds this size limit, it is truncated and marked with a star. For a limit of $k = 2$, the access path `a.next.prev.data` is reduced to `a.next.prev.*`[8]. This means that all sequences of field dereferences on base object `a` that start with `next.prev` match this access path. In other words, the set of referenced heap objects is conservatively over-approximated. With k-limiting in place, the set of possible access paths pointing to the same runtime object inside a program of finite size is finite and the analysis is guaranteed to never miss a potential sequence of dereferences for a tainted heap object. Thus, k-limiting is sound and space-bounded. On the other hand, k-limiting can lead to false positives. The reduction in the example from `a.next.prev.data` to `a.next.prev.*` makes the analysis oblivious to whether `a.next.prev.data` or `a.next.prev.foo` was the original reference, leading to a false positive in line 9. Note that this imprecision not only occurs with recursive data structures, but can always occur if the value of k is too low to capture the full depth of the data structure at hand. Recursive data structures only force the problem as they allow for infinite loops and would thus require an infinite k for precise modeling.

Note that recursive data structures are not only a theoretical problem, but occur frequently in practice. The `LinkedList` class in the Java Collections API, for instance, is a doubly linked list, i.e., every node in the list has a reference to its predecessor and to its successor. This makes the `LinkedList` code similar to the example from Listing 5. In the Java `TreeMap` implementation, every node contains references to its parent as well as to its left and right child. This is, again, a recursive data structure. Even worse, every reference to an inner class stored in a field of its outer class is a recursive data structure. When compiling an inner class, the Java compiler adds a field with the name `this$0` that points to the instance of the outer class with which the inner class was instantiated. An access path can loop between inner and outer class: `a.inner.this$0.inner.this$0...`.

The choice of k not only influences the precision of the analysis, but also its runtime. A k that is unnecessarily high wastes time and space by referring to individual elements that could also have been grouped together without any loss of precision. Consider the Java String library. If this string contains tainted data, it is usually sufficient for the taint to reference the string. Instead referencing the string's internal character array plus the length field of the string plus its hash as three individual access paths is, in most cases, wasteful, though sound. A simple solution would be to detect the maximum depth of all data structures in the program under analysis and use this value for k. As explained above, this approach, however fails for recursive data structures. Choosing a k value that is too low, on the other hand, leads to over-tainting, i.e., the taints referencing more heap objects than necessary. Beside the reduced precision, this can also increase the runtime of the analysis. More accesses to heap objects match the short access paths, and thus more new taints get derived. Every spurious match of an access path must be propagated through the program just like a correct one. In total, the number of propagated taint abstractions can increase significantly. In summary, while access paths have a conceptually simple correlation to the precision of the analysis, the impact of the access path length on performance and memory consumption is neither linear nor trivial and depends on the target program being analyzed. In Section 8.3, we evaluate the impact of increasing the access path length on the time and memory consumption of the data flow analysis in detail.

With these issues, k-limiting alone is not sufficient for a precise, yet efficient static data flow analysis tool. Deutsch [37] proposed *symbolic access paths* to deal with recursive data types. His analysis aims at detecting loops

[8] In the definition we use in this thesis, the base object does not count toward the maximum access path length. A length of `k` means an optional base object plus at most `k` fields.

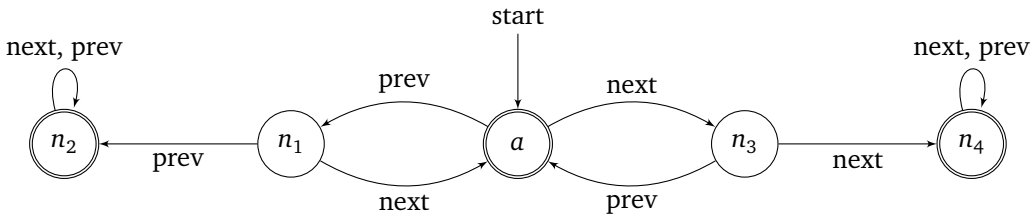


Figure 2: Access Graph for the Doubly-Linked List Recursive Data Structure

in access paths and reducing them to their canonic representation which is usually the one skipping the loop. Consider again the example from Listing 5. The access path of minimal length that references the tainted heap object is `a.data`. All field sequences `next.pred` in between can be left out without losing precision or soundness. These two additional field references only return to the very same object that would have been referenced without them. More formally, let `x` be a prefix of an arbitrary access path and let `y` and `z` be arbitrary sequences of field references. An access path `x.y.z` can be reduced to `x.z` if `y` only points back to `x`. In Deutsch's work, such an omittable access path fragment `y` is called a *Basis*. For the example in Listing 5, `next.prev` is such a basis `B`. The symbolic access path points to $a.(B)^n.data$.

In practice, whenever a new access path is created in the taint analysis, it must be reduced to its corresponding symbolic access path. This requires the analysis to check whether the new access path contains a basis or a sequence thereof. In general, a basis corresponds to a must-alias relationship. If `x.B` must-aliases with `x`, then `B` is a base. Deutsch's definition, however, is less precise: *if an access path π yields an object of type t when applied to an object of type t, then $\pi \in B_t^*$* (taken from [37]). Note that Deutsch refers to an access path as only the sequence of fields and thus applies access paths to base objects, instead of considering the access paths to consist of a base object and a sequence of field dereferences. More importantly, though, his definition of a basis is only based on types. A basis is defined for a type and can be stored in a constant lookup table. While this allows basis' to be precomputed for all types referenced in the target program to increase efficiency, it is also an imprecise approximation of the must-alias relationship. For the example in Listing 5, `next` would be a basis on its own as its return type is `A` again. The same holds for `prev`. This, however, can merge two access path that normally refer to different runtime objects into the same access path: `a.next.data` and `a.prev.data` both become $a.(B)^n.data$. If one is tainted and the other one is leaked, a false positive occurs.

Symbolic access paths are similar to *Access Graphs* as proposed by Khedker et al. [74]. This technique has already been used for modeling heap objects in a precise liveness analysis for the construction of points-to sets and callgraphs by Padhye [105]. An access graph represents a possibly infinite set of access paths, all of which are accepted by the regular language that is equivalent to the graph. Since the expressiveness of access graphs is limited to regular expressions, they cannot support unbounded counting. Consequently, the semantics of the doubly-linked list in the example cannot fully be represented. Normally, the object a can only be reached through access paths with an equal number of calls to `next` and `prev`. If more dereferences occur in one direction (next or prev) than in the other, the target might no longer be the same object. The program under analysis is free to use an arbitrary number of such dereferences. To over-approximate this memory structure, on would need an access graph with a single state and every operation (next or prev) stays in that state, effectively reducing `a.next.*` and `a.prev.*` both to `a.*`. Such a model is semantically equivalent to symbolic access paths. In general, access graphs are, however, more expressive than symbolic access paths, because they can exploit all patterns of pointer dereferences that can be modeled using regular expressions. Figure 2 shows an access graph that selectively over-approximates the heap structure imposed by the example data structure. It accepts sequences of `a.prev.next` or `a.next.prev` in which an operation is immediately followed by its counterpart. Additionally, as an over-approximation, it also accepts access paths that start with the same operation performed more than once such as `a.next.next.prev`. In other words, in such cases in which the automaton would be required to count, it simply accepts in states n_2 and n_4, respectively. Technically, the counting problem was unrolled for counting up to 1 operations. One can construct similar automata for any given finite limit k, which brings back an issue similar to the original one of k-limiting. Additionally, automatically inferring such complex models is a challenge on its own when given only the program code of the data structure and its operations.

Access Path Abstraction by Lerch et al. [81] is a different approach to the problem. The key idea is not propagate many different access paths that are similar, but to only propagate a single abstraction for all access paths that are rooted at the same variable. If two abstractions with the same base variable arrive at a control flow merge point,

they are joined together to the same abstraction which is then propagated onward, leading to much fewer taints being propagated through the program. Even more importantly, if analysis information computed for individual methods is summarized for later re-use, these summaries are applicable to more contexts. A summary for an access path rooted in `x` is applicable to all incoming access paths rooted in `x` instead of having to compute a new summary for every access path. To still retain full precision, the full access paths are reconstructed and matched when they are accessed. While this technique has the potential to avoid the two problems of (1) infinite access paths for recursive data structures, and (2) large numbers of access paths being propagated due to over-tainting, it requires significant changes to the taint engine. Whenever fields are read, the analysis must on-demand reconstruct the original access paths. Similarly, field writes, method summaries, loops, etc. need special treatment. In total, this approach requires significant changes to an existing taint analysis. Combining the approach with FlowDroid would be an interesting area of future work. Note that Lerch's work was presented after we published our work on FlowDroid.

Kanvar and Khedker provide a survey on different heap abstraction techniques known to literature [72]. They classify the techniques according to two criteria: heap modelling and summarization. In their terminology, heap modelling refers to the collection of a potentially unbounded set of concrete locations, and summarization refers to mapping these locations to a finite set of abstract objects. Heap models can be *store-based* or *storeless*. Access paths are storeless, because they only represent how memory can be accessed inside the program through a sequence of field dereferences. In a store-based model, on the other hand, heap locations are identified through their addresses, representing by the incoming pointers to these addresses. More simply, a store-based model is a heap graph in which every object A that holds a pointer to another object B corresponds to a directed edge from A to B, very much like access graphs. Similarly, access paths can be seen as (potentially unbounded) paths through the heap graph. K-limiting and referencing heap objects through their respective allocation sites are two examples of summarization techniques. Note that k-limiting is applicable to both store-based and storeless heap models. In the store-based case, it reduces the heap graph by removing nodes and inserting over-approximating edges that take the place of the pointers to the removed objects. In the storeless case, k-limiting works as described above, i.e., by truncating the access path and appending a wildcard operator instead. Another common summarization technique for store-based heap models is to identify a heap object by its allocation site. While this technique is simple and greatly reduces the number of distinct heap objects that an analysis must handle, it cannot properly deal with factory methods. All heap objects created in the same factory method would be considered the same heap object, which is not commonly the case.

3.4 The IFDS Framework

The FlowDroid data flow tracker which is central to the work presented in this thesis is based on the IFDS framework by Reps and Horwitz [117]. IFDS stands for a class of problems to which their framework is applicable: *inter-procedural*, *finite*, *distributive*, *subset*. We will now explain each of these conditions in detail and show that they are fulfilled for the problem of static data flow analysis. Firstly, problems must be defined over a finite domain (i.e., there is only a finite, enumerable set of distinct data flow abstractions). When using access paths of a finite length, this is trivially the case. There can be at most $O((n+m)^k)$ distinct access paths where n is the number of local variables, m is the number of fields across all classes in the target program, and k is the maximum access path length. Note that this is only a theoretical upper bound. In practice, there will be fewer taint abstractions, because not every arbitrary sequence of fields is a valid access path, i.e., a valid sequence of field dereferences according to the definitions of the respective classes.

Secondly, the problem must be distributive. If a certain node in the program' control flow graph is reachable on two different paths through the program, it must be possible to first compute the data flow abstractions along each path and then join them, and this must yield the same result as computing them together. Or, in other words, it must be irrelevant *when* separately-computed data flow abstractions are joined. Static data flow tracking works by propagating a set of taint abstractions. Unconditional taints are associated with the sources and these taints are then propagated over all successor nodes in the program's interprocedural control flow graph. When new taints are created, e.g., because a tainted variable is assigned to another variable, this simply creates a new taint abstraction for that second variable. In total, each statement is associated with a set of data flow abstractions denoting the access paths of all tainted data at that statement. It does not matter for which statement we first compute a new element for this set. Adding elements to sets is trivially distributive: It does not matter whether new taints are added to the set right away or kept separate to afterwards merge the partial sets. The last requirement, subset semantics, is also trivially fulfilled, because the merge operator of the analysis is set union on a lattice of sets of taint abstractions. The bottom of the lattice is the empty set, each added taint abstraction forms a new lattice element. Consequently, taint tracking meets all requirements and is an IFDS problem.

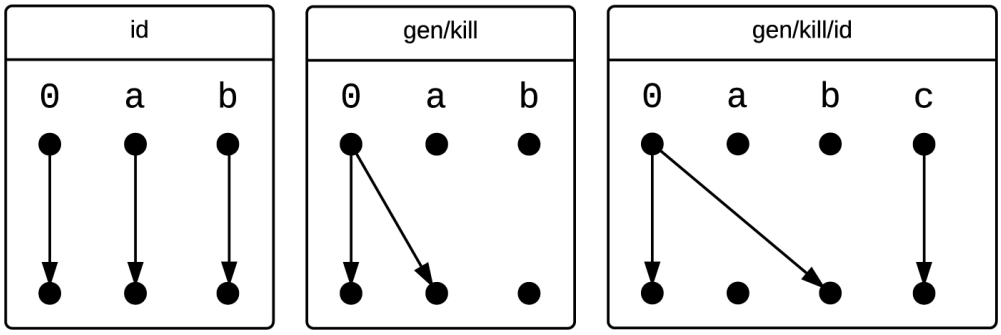


Figure 3: IFDS Flow Functions, Reproduced From [117]

Reps and Horwitz have defined a flow- and context-sensitive framework for solving problems that meet the above criteria. Naem et al. [100] have contributed practical extensions for efficiently applying the IFDS framework to problems defined on real-world programs. Bodden has implemented Heros, an IFDS solver on top of Soot [23], which we initially used for FlowDroid, before switching to our own, more efficient solver implementation described in Section 4.12. All these solvers share the same general concepts. They start with an inter-procedural control flow graph of the target program and expand it to an *exploded supergraph*, i.e., each node in the control flow graph is turned into one node per possible abstraction. A fact f_1 holds at a statement s_1 if the respective node $\langle s_1, f_1 \rangle$ is reachable from the start node $\langle s_0, 0 \rangle$. The statement s_0 is usually the first statement in the program. The fact 0 is a special fact that holds unconditionally. The data flow problem is thus reduced to the problem of generating the exploded supergraph and then computing reachability relationships in it. Note that in more efficient (albeit non-standard) implementations such as FlowDroid, the start nodes $\langle s_0, 0 \rangle$ are created directly at the sources, not at the start of the program.

To create the exploded supergraph, the data flow functions must be encoded as edges between the nodes of possible facts at program statements. For two control-flow connected successor statements s_1 and s_2, node $\langle s_1, f_1 \rangle$ is connected to a node $\langle s_2, f_2 \rangle$ if and only if the fact f_2 holds at statement s_2 given that the fact f_1 previously held at statement s_1. Generating such edges is the purpose of the flow functions. Figure 3 (reproduced from [117]) shows graph encodings for typical flow functions. The function on the very left side is an `id` function that retains all incoming flow facts. The function in the middle unconditionally generates a new fact for `a` (`gen` function) and discards the incoming fact for `b` (`kill` function). The `kill` part is modeled by the missing arrow. The function on the right side combines `gen`, `kill`, and `id`. It kills `a`, unconditionally creates `b`, and retains `c`. Note that the flow functions used in static taint tracking depend on the statements they connect, i.e., they are not what is usually called *separable*. A separable flow function would only depend on the incoming data flow fact, but not on the current statement. In taint tracking, a statement that overwrites a certain variable, would for instance be translated to a `kill` flow function for the repsective taint abstraction, while a statement that does not access that variable at all would be translated into an `id` function, leaving the taint abstraction as it is.

We generally distinguish four different types of flow functions depending on the type of the current statement:

- **Normal flow function** A normal flow function is applied when the current statement is neither a call site, nor a return site. Common cases of such statement are assignments and conditionals.

- **Call flow function** A call flow function models a method call and is applied at call sites only. It it responsible for mapping actual call arguments in the caller to formal method parameters in the callee. If the call is to an instance method, it must also map the base object of the call to the `this` object of the callee. It is not responsible for data flow facts that shall not be mapped into the callee, but passed on inside the caller.

- **Return flow function** A return flow function is applied at the exit node of a method. This can be a `return` statement or a statement at which an uncaught exception is thrown, because the control flow will then also leave the current method. The return flow function performs the inverse operations of the call flow functions, i.e., it maps back the parameters and (if applicable) the base object and the return value from the callee into the caller. All flow facts that are no longer valid because they have no corresponding fact inside the callee are dropped.

- **Call-to-return flow function** A call-to-return flow function is applied at call sites. It models flow facts that skip the callee. Instead of mapping the facts into the callee as in the call flow function, it maps them to the

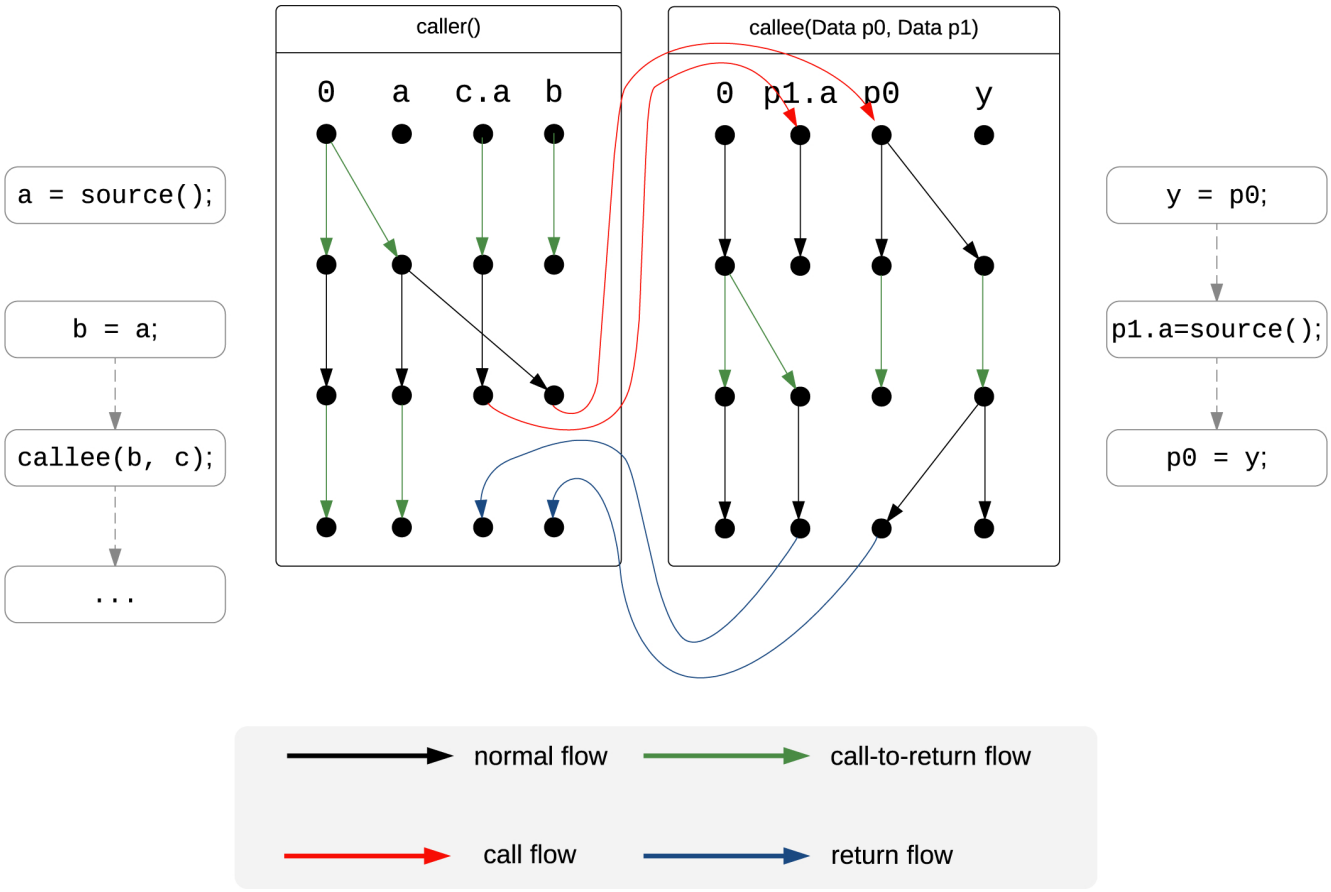


Figure 4: Four Types of IFDS Flow Functions

successor statement inside the caller. A common use case is to preserve facts about variables that are not in the scope of the callee.

In Figure 4 we give an example of how the four different flow function types can be defined for a simple static data flow analysis. The analysis starts in the `caller()` function. We assume that the analysis considers sources as black boxes. Consequently, calls to a source method are processed in the call-to-return flow function (green edges). This links the variable `a` to the unconditionally tainted zero fact. The next line copies the taint state of variable `a` to variable `b`. Afterwards, the call maps variables `b` and `c` into the context of method `callee` (red edges). Note that we only included `c.a` in the figure, because neither `c` itself nor any other field in it ever receives any taint. Consequently, all edges on `c` would have been identity edges which we omitted for not cluttering the presentation. Inside the `callee()` method, new unconditionally-tainted data is assigned to `p1.a`. When returning from `callee()`, the respective data flow fact must be mapped back into the caller (green edges) to make sure that the taint on the field is not lost. In principle, all IFDS-based data flow analysis are built using similar semantics and definitions. Note that a precise real-world taint analysis such as FlowDroid is more complex, though, because it also needs to handle issues such as aliasing.

The original IFDS paper proposed to first create the complete exploded supergraph and then compute reachability based on it. While this is conceptually simple, it is inefficient in practice, because it would require flow functions to be enumerated even for statements that are never actually reached by a data flow fact. This can happen because of unreachable methods or because even in reachable methods, not all possible access paths are actually tainted at some point. Therefore, we compute the exploded supergraph on-demand, as explained in [100]. A flow function is created only when it is actually required. In practice, the implementation runs a fixed-point iteration on a taint set. It takes a taint at a statement $\langle s_1, f_1 \rangle$ from this set and computes the respective flow function to obtain the new set element $\langle s_2, f_2 \rangle$. The iteration stops when no new entries are obtained, i.e., the set has reached its fixed point.

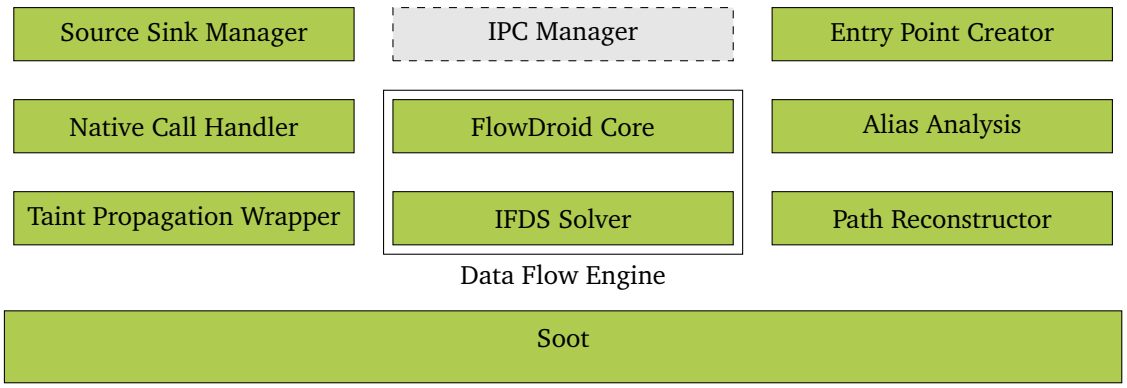


Figure 5: FlowDroid's Architecture

4 Precise Static Data Flow Analysis: FlowDroid

开始介绍FlowDroid这个taint tracer的具体实现

FlowDroid [9] is a static data flow analysis tool for Android apps and Java programs. Given a specification of sources and sinks, it enumerates all possible data flows between any of these sources and any of these sinks. In this section, we explain in detail how FlowDroid works, what its features and limitations are, and how it can be used as an extensible framework for further research in the field of static data flow analysis. The main goal of our work is to produce a taint tracer that is effective and efficient for finding privacy leaks in real-world scenarios. Our description follows the architecture of the tool and explains the various components from which it is built. Section 4.16 will later map the components onto a temporal workflow and show in which order the components are applied to conduct the data flow analysis.

4.1 Architecture

Figure 5 shows an overview over the various components of FlowDroid. All components are based on the Soot compiler optimization framework [78]. This framework is responsible for converting Java bytecode into the Jimple intermediate representation [137]. As explained in Section 3.1, Jimple is much easier to analyze for a static analysis tool than the Java bytecode language. The FlowDroid data flow tracker is based on the IFDS framework by Reps and Horwitz [117] explained in Section 3.4. Expressing the data flow problem as an IFDS problem allows FlowDroid to build on existing techniques for propagating taint abstractions over an inter-procedural control flow graph. From IFDS, FlowDroid automatically inherits context-sensitivity and flow-sensitivity. By using access paths as the primary taint abstraction propagated by the IFDS solver, FlowDroid ensures object-sensitivity. The implementation of the data flow problem on the Jimple IR is called the *FlowDroid Core*. Together, the IFDS solver and the FlowDroid Core form the *Data Flow Engine*. All properties of the data flow analysis that differ depending on the concrete target platform (Android or Java) or problem configuration (e.g., which sources and sinks to consider) have been encapsulated in interfaces outside of the data flow engine. These are the other components in Figure 5[9] which will be explained in the remainder of this Section:

- **Source Sink Manager** Responsible for deciding which statement is a source or a sink, see Section 4.3
- **Alias Analysis** Exchangable interface to plug different alias analyses into FlowDroid. See Section 4.8.
- **Taint Propagation Wrapper** Provides optional external models for shortcutting taint propagations over calls to libraries. See Section 4.9.
- **Native Call Handler** Provides external models for propagating taints over calls to native methods. See Section 4.10.
- **Path Reconstructor** Component for extracting source-to-sink connections and optionally data flow paths from the taint propagation graph. See Section 4.13.

[9] Some interfaces such as the one for the alias analysis are usually only used with the default implementation

- **IPC Manager** Optional component that can be used by tools that extend FlowDroid with support for inter-process communication (such as inter-component and inter-app communication on Android). The default implementation in FlowDroid is a stub.

- **Entry Point Creator** Component for creating a dummy entry point for callgraph construction if the target does not provide a readily usable (i.e., a single, static, public) entry point. See Section 4.15.

This separation between data flow engine and customizable components allows FlowDroid to be applied to various problems without changing the implementation of the data flow engine. If not specified otherwise, default implementations are used which model the semantics of traditional Java programs. One major focus of our work on FlowDroid is to provide precise and efficient alternate implementations that capture the semantics of the Android operating system and framework. The platform-specific alternate implementations of these interfaces that we created for Android are explained in Section 5. Regardless of the FlowDroid-specific interfaces, the data flow tracker is, however, always based on Soot and it's Jimple intermediate representation. Therefore, the set of targets that can be analyzed is limited to what can be expressed in Jimple. For reading in Android apps (which use a different bytecode format than traditional Java programs), we use Soot's Dexpler component [18]. Analyzing platforms other than Java and Android (and thus converting the respective code to the Jimple IR) is discussed in Section 9.

The main abstraction that is propagated across statements in FlowDroid is access paths which are explained in detail in Section 3.3. Recall that an access path is a sequence of field dereferences, e,g. `obj.fld1.fld2` references the object that is reachable by taking the base object `obj` and then first accessing field `fld1`, and, in the object obtained from this field, accessing the field `fld2`. While the access path alone would be sufficient to perform basic data flow tracking [10], additional information is stored along with each access path to support certain features. For example, FlowDroid keeps track of precise type information for the base object of the access path and each dereferenced field in order to prune impossible callgraph edges. These additional fields inside the access path are explained in the respective feature's section.

4.2 The FlowDroid Core

The FlowDroid core is responsible for the basic mandatory features of the data flow tracker. It formulates the interprocedural taint propagation as an IFDS problem. The core handles cases such as simple assignments, method calls, and method returns. In this section, we will describe the core's IFDS flow functions (normal, call, return, call-to-return) semi-formally. For not bloating the basic data flow tracking with more complex or optional features, some aspects (arrays, implicit flows, sources and sinks, etc.) were factored out into the *rule engine*. Each rule inside the rule engine can provide more rules to the four basic data flow functions, e.g., add an additional rule on how to process array accesses, which is not handled by the core. The details of the rule engine are discussed in Section 4.2.7. This section focuses on the core. Therefore, the features implemented in rules do not affect the flow functions discussed in this section. Also note that the core can only transfer, preserve or kill taints, but not unconditionally generate new ones; the latter is handled in the source propagation rule.

Note that FlowDroid computes flow functions pointwise. Without loss of generality, we therefore assume that there is only a single incoming taint abstraction on which a flow function needs to be evaluated. This greatly simplifies our formalization. For each incoming taint abstraction, the flow function can generate an arbitrary number of outgoing abstractions which are then, again, propagated onwards individually. If a single statement is reached by multiple incoming data flow facts, the respective flow function is evaluated once for each incoming fact. Each outgoing fact contributes to the set of facts that hold at the statement, but is, following the scheme of pointwise propagation, propagated onwards independently.

4.2.1 Normal Flow Function

Normal flow functions are applied at all statements that are neither calls nor returns. The core must thus only handle assignments. Keep in mind that optional functionality such as implicit flow handling (for which one would need to define a normal flow function for conditionals) is part of a separate rule and not of the core. Further note that the right side of an assignment in a normal flow function is never a method call. Assignments that contain calls

[10] It would also need a reference to the source from which the tainted data was originally derived.

```
1  public class Container {
2    private Object fld;
3
4    public Object get() {
5      return this.fld;
6    }
7
8    public void test() {
9      Container c = new Container();
10     c.fld = source();
11     new UserCode().leak(c);
12   }
13 }
```

```
14 public class UserCode {
15
16   public void leak(Container c) {
17     Object o = c.get();
18     sink(o);
19   }
20
21 }
```

Listing 6: Visibility of Access Paths

are handled by the call flow function. For an assignment statement $s \in Stmt$ with the structure $x.f^n = y.g^m$ with $n, m \in \{0, 1\}$ the following rules apply[11]:

- For an incoming taint $T = y.g^m.h^k$ with $k \in \mathbb{N}_{\geq 0}$, the new taint set is $\{T, x.f.h^k\}$. Note that h^k denotes an arbitrary, potentially empty suffix to the access path and not a k-times repetition of the same field dereference.
- For an incoming taint $T = y.*$, the new taint set is $\{T, x.f.*\}$.
- For an incoming taint $T = x.f^n.g^k$ with $k \geq 0$, the new taint set is $\emptyset$. If the field that is overwritten is a prefix of the incoming tainted access path, this taint is killed.

Note that the first and third rule can also have the star wildcard appended to both sides of the rule. In general, if a taint rule applies without the star, it also applies when appendiung a star to both the incoming and the outgoing taint. All taints to which none of the above rules apply are copied over as-is.

4.2.2 Call Flow Function

The call flow function is applied to all statements that contain a method call, including assignments with a method invocation on their right side. However, note that the left side of such an assignment can only be tainted when processing the return flow function, because only then, the effect of the callee on the incoming taint is known. When discussing the call flow function we can, therefore, without any loss in generality, omit the case of an assignment. For a call statement $s \in Stmt$ with the structure $o.m(a_0, ..., a_n)$ with $n \in \mathbb{N}$, the following rules apply:

- For an incoming taint $T = a_i.h^k$ with $k \in \mathbb{N}_{\geq 0}, 0 \leq i \leq n$, the new taint set is $\{p_i.h^k\}$ where p_i is the variable that stores the i$^{\text{th}}$ formal parameter in the callee. Note that this rule also applies to $T' = a_i.*$ becoming $p_i.*$ by setting $k = 0$ in the rule and applying the general principle that stars can always safely be added on both sides of a rule.
- For an incoming taint $T = o.h^k$ with $k \in \mathbb{N}_{>0}, 0 \leq i \leq n$, the new taint set is $\{this_i.h^k\}$ where `this` is the *this*-reference inside the non-static callee.
- For an incoming taint $T = S.h^k$ with $k \in \mathbb{N}_{\geq 0}$ where S is a static field, the new taint set is $\{T\}$.

Again, note that h^k denotes an arbitrary, potentially empty suffix to the access path and not a k-times repetition of the same field dereference. Unlike the case of the normal flow function, all incoming taints for which there is no explicit propagation rule are discarded as they are assumed not to be accessible from within the scope of the callee. However, beware that an access path being in scope in a callee does not necessarily imply that is visible or

[11] There is at most one field derefrerence on either side of the assignment and there cannot be a field access on both sides at the same time according to the specification of the Jimple IR.

accessible from the callee as well. In the example in Listing 6, the tainted data is read from the source in Line 10 and stored into a private field of the container object. This leads to a tainted access path `c.fld`. In Line 11, this taint is passed into the callee `UserCode.leak()`. Inside there, `c.fld` is in scope, though the private field `fld` is not visible. It would, however, lead to a false negative to prune this "invisible" taint, because `UserCode.leak()` in turn calls the `get()` method on the container object in Line 17 which *can* read the private field because it is in the same class. Therefore, access paths must be propagated into callees regardless of the respective fields' access modifiers.

Some callees require special handling. The above rules implicitly assume that the method reference at the call site matches the callee's declarations, i.e., the number of call arguments is equal to the number of formal parameters in the callee. For proper callgraph edges, this is an invariant. Soot's SPARK callgraph, however, includes additional pseudo edges that model special cases such as threads and that violate this invariant. In Java, one possibility for starting a thread is calling `ThreadPoolExecutor.execute()` and passing it an instance of a custom class that implements the `java.lang.Runnable` interface. This custom class then implements the `run()` method in which the thread's code, i.e., the code to be executed concurrently, is placed. It is the duty of the Java Runtime (or the Android framework in the case of Android) to actually start the thread on the OS level and invoke the `run()` method in the context of the new thread. This operating-system specific handling is implemented in native code. Consequently, even when the full Java or Android library is analyzed together with the target program, there is no call edge from `ThreadPoolExecutor.execute()` to `Runnable.run()` that would be visible to the callgraph algorithm, effectively making the `run()` method unreachable. To circumvent this problem, the SPARK callgraph algorithm fakes an edge from all call sites of `ThreadPoolExecutor.execute()` to `Runnable.run()`, effectively injecting the respective implementation of `Runnable.run()` as the callee for the `ThreadPoolExecutor.execute()` call sites. Semantically, this simulates that the thread code is "inlined", i.e., synchronously executed at the respective call site of `ThreadPoolExecutor.execute()`. In the call flow function of the data flow analysis, such call special edges must be handled separately. While `ThreadPoolExecutor.execute()` takes one parameter (namely the instance of `java.lang.Runnable` containing the thread code), the `run()` method that is finally called is parameterless. The call flow function must therefore not try to map any parameters in this case. Similar special-casing is required for the `doPrivileged` methods of the Java code security infrastructure.

4.2.3 Return Flow Function

The purpose of the return flow function is to map taints that are valid inside a callee back into the original caller when the control flow returns from callee to caller. It can be seen as the inverse of the call flow function. Similar to the call flow function, all taints to which no explicit propagation rule applies are discarded as they are assumed to only be in scope inside the callee, but not the caller. Regardless of the return statement in the callee, the following rules apply given an original call site $s \in Stmt$ with the structure $o.m(a_0, ..., a_n)$ where $n \in \mathbb{N}$. We will discuss call sites that contain assignments later on.

- For an incoming taint $T = p_i.h^k$ with $k \in \mathbb{N}_{>0}, 0 \leq i \leq n$, the new taint set is $\{a_i.h^k\}$ where p_i is the variable that stores the i$^{\text{th}}$ formal parameter in the callee.
- For an incoming taint $T = p_i.*$ with $k \in \mathbb{N}_{>0}, 0 \leq i \leq n$, the new taint set is $\{a_i.*\}$
- For an incoming taint $T = this.*$ with $k \in \mathbb{N}_{>0}, 0 \leq i \leq n$, the new taint set is $\{o_i.*\}$ where `this` is the *this*-reference inside the non-static callee.
- For an incoming taint $T = S.h^k$ with $k \in \mathbb{N}_{>0}$ where S is a static field, the new taint set is $\{T\}$.

It is important to note that the first rule in the call flow function has its equivalent in the *two* first rules stated above. In the first rule, an access path the starts with $\mathtt{p_i}$, but is *strictly longer* (h^k with $k > 0$, not $k \geq 0$ as in the call flow function) is propagated back. Additionally, if the access path ends at the parameter, but has the wildcard ($\mathtt{p_i}.*$), it is also propagated back. If the access path only taints the parameter, but not a field inside it (rule 1 with $k = 0$), the taint is discarded. In that case, the parameter value as a whole was overwritten inside the callee. According to the Java language semantics, the caller will still carry on with the old value when the callee returns. For primitive values and strings, the propagation rules can be simplified. Primitives do contain fields that can get tainted inside the callee and strings are immutable by definition. Therefore, taints on variables of these types can never be propagated back to the caller through parameters on a function call.

Still, these two rules are an over-approximation when dealing with an intermediate language such as Jimple that does conform to the static single assignment (SSA) form. In other words, variables in Jimple can be re-used

```
1  public void main() {
2    Container c1 = new Container();
3    Container c2 = new Container();
4    callee(c1, c2);
5    leak(c1);
6    leak(c2);
7  }
8  private void callee(
9      Container d1, Container d2) {
10   d1 = source();
11   source(d2);
12 }
```

Listing 7: Ambiguity of Overwritten Access Paths

```
1  void callee(Container c1, Container
       c2) {
2    leak(c1);
3
4    c1 = source();
5    source(c2);
6  }
```

Listing 8: Local Parameter Variables and Local Splitting

```
1  void callee(Container c1, Container c2) {
2    if (Math.random() < 0.5)
3      c1 = source();
4    source(c2);
5  }
```

Listing 9: Local Splitting Usage Ambiguity

at any time, only requiring that all uses of the variable are compatible with the same static type. Consider the example in Listing 7. The `main()` method creates two new instances of the `Container` class, none of which are initially tainted. It passes both of them to the `callee()` method. Inside `callee()`, the variable holding the first parameter is overwritten with tainted data from the source, tainting `d1.*`. This value should not be visible inside the caller. After line 10, the variable `d1` no longer points to the original object that was passed into `callee()`. As the caller `main()` always continues with the old (untainted) object, no taint should be propagated backwards. On the other hand, the second parameter `d2` is not overwritten, but the very same object that was passed into `callee` is passed on to (another overload of) the source method. Assume that this method taints something inside `d2`, but is is modeled using a taint on `d2.*` as well as an over-approximation[12]. In that case, the taint must be passed back to the caller `main()`. For the FlowDroid core, the two taints `d1` and `d2` are, however, indistinguishable without any further information about the concrete behavior of the source. Both taint a parameter variable and all of its fields. With the rules above, both will be propagated back into the caller, producing a false positive in line 5. Choosing to not propagate back either would lead to a false negative in line 6 which, from a security / privacy analysis standpoint, is a worse decision to make.

This ambiguity could be resolved by applying FlowDroid to an SSA-type intermediate language such as Shimple [136] instead of Jimple which allows variable redefinitions. Another approach would be to use Soot's local variable splitter that identifies unrelated uses of a variable and then splits these uses so that each of them operates on a separate variable. At first, this appears to be an easy solution that can be done as pre-processing step before executing the data flow solver and thus does not require any changes to the FlowDroid core. In the example in Listing 9, the local splitter would create a new variable `c1$1` for the definition in line 4 instead of overwriting `c1`. When method `callee` returns, this new variable is clearly independent from the parameter variable. Thus, the analysis can distinguish that the taint on `c1` must not be mapped back into the caller, while the taint on `c2`, which is still the original parameter variable, has to be mapped back. Still, this approach does not fully solve the problem. Consider the code example in Listing 9. In this slightly modified case, the original parameter value of `c1` is only overwritten depending on some condition. For the local splitter, there are no distinct uses of `c1`. Both the parameter variable `c1` and the definition at line 3 merge into the same variable `c1`. Consequently, the ambiguity of whether `c1` must be mapped back into the caller when the method `callee` returns, remains. We note that depending on the front-end with which Soot reads in the target program (Java bytecode, Java source code, Android bytecode,

[12] As explained in Section 4.3, source methods are modeled using user-defined *Source Sink Managers* that receive method calls and can create arbitrary taints for them. This means the data flow core has not control over the definition or level of granularity of the sources. Implementing an over-approximating source sink manager that taints the whole object that is passed as a parameter to a source method is a legitimate design choice.

etc.), it may already apply the local splitter during import phase, making this partial solution an implicit part of the FlowDroid analysis.

To completely solve the problem without changing the IR, additional bookkeeping would be necessary. For each nested call site, FlowDroid would have to keep track whether the variable was overwritten or tainted (i.e., passed to a source method as a parameter). More generally, the return site needs to know not only the taint state of the parameter variable, but also how this taint was created. If we have nested calls, this information must be available for each call level from which the control flow (and thus the taint tracking) returns. Conceptually, this is similar to checking whether the parameter variable at the beginning of the callee may-alias with the parameter variable at the return site. Such a may-alias relationship only exists if there is at least one path through the callee (more precisely: from the start of the callee to the current return site) on which the variable is not overwritten. Conceptually, FlowDroid's existing alias infrastructure can deliver such precise context- and flow-sensitive alias information. As we explain in Section 4.8.4, this implementation is, however, implemented as a second IFDS problem that runs largely independently in its own solver instance. More precisely, aliases are aways computed asynchronously; the data flow solver only triggers the alias query, and immediately continues. When the alias query finally completes, it creates new taints from the discovered aliases which is independent from the flow function that originally triggered the alias analysis. With this design, aliases can only be used to generate new leaks, but not for influencing how an existing taint shall be processed. The decision of whether to map back a taint or not in the return flow function cannot be based upon the outcome of such an alias query without synchronizing the alias solver with the data flow solver, which is likely to incur a high performance penalty. In this work, we decided against this additional overhead and rather chose to implement an approximation. If the parameter variable was never assigned to in the callee, we propagate the taint back to the caller, otherwise we discard it. This design decision is based on the assumption that source sink manager implementations that taint parameters are uncommon, because such sources are uncommon in Android and Java. In the SuSi project [109] that analyzed the complete Android source code for sources and sinks using machine learning, no such sink was discovered.

In the description above, we have assumed that the call site to which the call flow function is applied is not part of an assignment. If the callee, however, returns a value and this value is used in an assignment, there is an additional rule. Assume that the call site has the structure $x = o.m(a_0, ..., a_n)$ where $n \in \mathbb{N}$ and that the exit site is of the form `return r`[13].

- For an incoming taint $T = r.h^k$ with $k \in \mathbb{N}_{\geq 0}$, the new taint set is $\{x.h^k\}$.

The Jimple intermediate representation does not allow a field reference on the left side of an assignment that has an invocation on its right side. Therefore, this case need not be handled and we can safely assume that left-hand side `x` is a local. The same applies for field references in the argument list of the call site, i.e., all arguments are either locals or constants. However, there is one special case to be handled: the left side of the assignment can reference the same variable as a parameter. In the example in Listing 10 in Line 4, the variable `val` is an argument to the call to `callee()`, but also receives the return value of that method. In Line 10, sensitive data is written into `c.fld`. If a snapshot of the program state were taken after executing this line, `val.fld` in method `main()` would also be tainted. When the control flow returns from `callee()`, the variable `val` is, however, overwritten with a new object that does not contain tainted data. Therefore, no leak occurs. FlowDroid models this special case explicitly in the return flow function and does not apply the first group of rules (parameter and base object handling) if they would taint an access path that starts with the variable that receives the return value.

However, this rule only applies under the assumption that the callee was processed completely before returning to the caller. This is not necessarily the case for exceptional control flow. Assume that the callee throws an exception in Line 13. In this case, the parameter value is already overwritten, but no return value is ever created and the assignment on Line 4 never happens. Consequently, the value `val.fld` never changes in the callee. FlowDroid models this behavior by only applying the assignment precedence explained above if the return control flow is not exceptional. If it is exceptional, taints on parameters and the base object are mapped back, even if the respective variable is overwritten.

4.2.4 Call-to-Return Flow Function

The call-to-return flow function is applied to all call sites and models the flow of facts that skip the callee and directly proceed at the return site inside the caller. This mainly applies to taints that are in scope in the caller

[13] We use the definition that an *exit site* is the last executed statement inside the callee, and that the *return site* is the statement at which the execution continues in the caller after returning from the callee

```
1  void main() {
2    Container val = new Container();
3    try {
4      val = callee(val);
5    finally {
6      leak(val.fld);
7    }
8  }
```

```
9  Container callee(Container c) {
10   c.fld = source();
11
12   // Create and return new object
13   Container c2 = new Container();
14   c2.fld = null;
15   return c2;
16 }
```

Listing 10: Parameters and Return Values

and thus need to be preserved on the caller side, but that are not in scope inside the callee and thus cannot be propagated into the caller and back again. For a call statement $s \in Stmt$ with the structure $o.m(a_0, ..., a_n)$ with $n \in \mathbb{N}$, the following rules apply:

- For an incoming taint $T = x.h^k$ with $k \in \mathbb{N}_{\geq 0} \wedge \forall p_i, 0 \leq i \leq n : p_i \neq x \wedge x \neq o$ and x not being a static field, the new taint set is $\{T\}$.
- For an incoming taint $T = p_i.h^k$ with $k = 0, 0 \leq i \leq n$, the new taint set is $\{T\}$

All tainits to which none of the above rules apply are killed. This is to enforce that no taints that must be propagated through the callee (to be potentially killed inside the callee or one of its subsequent callees) are kept alive in caller regardless of the callee. The first rule propagates those taints that are not in scope in the callee if the taint is neither on a parameter nor on the base object of the virtual method call. For static method calls, the check on the base object is not performed. Note that static fields are always propagated through callees, because they are always in scope in the whole program. The second rule is special as it preserves taints on values that are passed as parameters, but only reference the parameter as such, not any fields inside it. As mentioned for the return flow rule, such taints cannot be changed inside the callee, so these taints are only propagated into the callee (in case the taint is passed into a sink there), but not back out again. For not losing the taint on the caller side, it is preserved via the call-to-return flow function.

4.2.5 Optimizations

The rules explained above can be inefficient for deep call hierarchies. Assume a taint on `this.x`. In this case, for every call to some method `this.m()`, the taint is propagated into the callee, over all statements inside the callee, into all of the callee's callees, and finally back to the original caller even if field `this.x` is never read in any of those methods. This not only induces the effort to process this taint on each statement in each of these methods. Also recall that the flow-sensitive IFDS solver must, for each statement in the program, store all data flow facts that hold at the respective statement. If the analysis must store the field taint for all statements in the whole call tree, this introduces unnecessary memory consumption. The problem is even bigger for static fields, because a static field is always in scope. Once a taint on a static field has been created, it will recursively be passed into all callees that are subsequently being called in the control flow, regardless of whether this taint is relevant to any of them or not.

To circumvent this problem, FlowDroid applies a quick pre-analysis to check whether the first field in the access path is read in any of the transitive callees of a given method call. If this is not the case, the taint is propagated over the call-to-return edge and thus bypasses the call instead of being propagated into the callee, all the way through it, and back out again. Note that the result of this check is a context-insensitive property of the callee, i.e., can efficiently be cached and must only be done once per method and not once per call site or context. Note that we only apply this optimization to taints on static fields at the moment. Access paths that reference non-static fields are only propagated into callees if their base object appears in the call as the base object of the callee or inside the argument list of the call. Conservatively over-approximating whether an incoming access path $\texttt{a}_0\texttt{.f}$ can be read in the callee (assuming that a_0 is the first argument of the call) would mean checking whether p_0, the variable that receives the first formal parameter in the callee, is accessed. If this is the case, an alias could be created for later reading out the field even without synactically accessing $\texttt{p}_0\texttt{.f}$. However, the case in which a parameter of a method is completely unused is rather rare, so such a pre-check does not increase performance in practice and can even slow down the analysis due to the additional computational cost of the check.

4.2.6 Callgraph Considerations

The IFDS solver propagates taint abstractions along an interprocedural control flow graph. For building such a graph, one first needs to construct a callgraph from which the interprocedural edges are taken (mappings between call site and callee as well as between exit sites in callees and return sites in callers). To be able to evaluate the tradeoffs between callgraph accuracy and callgraph computation time, FlowDroid supports multiple callgraph algorithms: RTA, CHA, VTA, and SPARK. Computing a precise callgraph with, e.g., SPARK takes more time than creating a VTA callgraph. On the other hand, a VTA callgraph usually has many more (spurious) edges along which data flow information then needs to be propagated. Due to this effect, the time saved when computing a less precise callgraph can easily be spent again several times during the taint propagation on large programs. The callgraph computation time only needs to be invested once, but the spurious edges cause unnecessary propagations on every taint abstraction that reaches them. In our experiments, we found the latter effect to greatly outweigh the time saved during callgraph construction. Analyses that completed in about 80 seconds with about 9 GB of memory when using SPARK were no longer completed with CHA even when given several hours and 100 GB of memory. Therefore, FlowDroid uses the Soot's SPARK callgraph algorithm [84] with full precision by default.

Traditionally, this callgraph is computed upfront using existing algorithms such as Soot's built-in SPARK algorithm [84]. Unfortunately, these algorithms compute the potential callees for each call site in the whole program, regardless of whether tainted data will ever be tracked across the respective call edge or not. Therefore, it can be advantageous to compute the callgraph on demand. Such on-demand analyses exist for callgraph construction [2] as well as for pointer analysis [128] which is an important component of precise callgraph construction. While it has been shown that on-demand computation can significantly lower the computational cost of computation in the average case (i.e., for an average number of nodes being queried), they do not easily integrate with Soot's class and method loading infrastructure. In Soot, classes and methods are only loaded when needed. Even then, only signatures are usually loaded unless more information such as the method bodies is actually needed and requested by an analysis. Therefore, an on-demand analysis only actually improves the performance in practice if it also reduces the set of methods whose bodies need to be loaded. In other words, the on-demand analysis must not assume that it can access all method bodies in the program. Otherwise, the computation time of the callgraph itself is shadowed by the on-demand method loading that happens in both cases (on-demand callgraph or classic upfront computation) and only a marginal improvement can be measured. A special challenge lies with invocations of interface methods as shown in Listing 11. The call to the interface method in Line 14 can either be resolved by taking all methods `handle()` in all implementors of the `IMyInterface` interface, which would be imprecise, or by scanning the call hierarchy of the `test()` method to find all possible assignments to field `this.intf`. In the latter case, this would yield the possible concrete types of the `base` variable on Line 14. In terms of complexity, such a scan, would, however, be a full backward slice. In Lines 2 and 7, factory methods are called to instantiate the interface. For a obtaining precise type information, these methods would need to be analyzed as well. In total, an on-demand analysis that precisely handled interface invocations needs to traverse (and, consequently, load) many methods inside the program. Even if the actual analysis effort per method is low, it is hard to achieve a substantial overall performance gain, because the method loading time is the single most important dominating factor. After a few initial experiments, we therefore chose to leave the issue of on-demand callgraph construction for a Soot-based data flow analysis to future work.

Another limitation of the SPARK callgraph algorithm is that it is context-insensitive. This can lead to spurious call edges and thus potentially data flow edges. In practice, we have found that the effect of this imprecision on the result of the data flow analysis, i.e., the leaks reported, is very limited. For most apps, not a single false positive is reported in the end. Even without false reports, there can, however, be an impact on performance. For every outgoing call edge at a given call site, the call flow function must be computed. If the result of this function is not the empty set, these new taints must be propagated through the whole call tree (i.e., the callee and all of its transitive callees). The performance impact is the more substantial, the more call edges there are at a umbiguous call site, and the larger the call tree of any of those false callees is. Consider the example in Listing 12. Method `main()` starts a new thread and passes tainted data to it. In a simplified version of the `java.lang.Thread` class, we can assume that `Runnable` implementation passed to the constructor is stored in a field. Calls to the `start()` method then create a new OS-level thread and invoke the `run()` method of the `Runnable` stored in the field. To precise identify the callee of this call to `Runnable.run()` inside the `Thread.start()` method, the callgraph analysis would need to differentiate the different instances of the `Thread` class and their different field values. SPARK, however, cannot distinguish such different contexts. Instead, the call to `Thread.start()` has outgoing edges to `Runnable.run()`, more precisely, to each `Runnable.run()` implementation that is ever passed to a constructor of the `Thread` class. In the example, both calls to `Thread.start()` are assumed to reach both `Runnable` implementations, leading to

```
1  void onCreate() {
2    this.intf = factory();
3    test();
4  }
5
6  void onPause() {
7    this.intf = otherFactory();
8    test();
9  }
10
11 void test() {
12   String s = source();
13   IMyInterface base = this.intf;
14   base.handle(s);
15 }
```

Listing 11: Interface Invocation and Callgraphs

```
1  void main() {
2    Thread t1 = new Thread(new Runnable() {
3      void run() {
4        // Thread code
5      }
6    });
7
8    t1.start();
9    Thread t2 = new Thread(new Runnable() {
10     void run() {
11       // Thread code
12     }
13   });
14   t2.start();
15 }
```

Listing 12: Imprecisions due to Context-Insensitive Callgraph

cross-references between `t1` and `t2`. This imprecision is due to the missing context, the callgraph algorithm cannot link the call site to any information limiting the possible states of the internal receiver field of the `Thread` class. For programs with a large number of threads, this imprecision can significantly incease the callgraph. Fortunately, most Java programs and Android apps do not use a very large number of threads. Note that the problem does not arise for the second overload of `Thread.start()` which directly takes the `Runnable` as an argument. In this case, SPARK can directly generate an edge from the call site to the `run()` method of the object passed using the parameter and the context of the call is irrelevant.

Soot supports the demand-driven context-sensitive points-to analysis by Sridharan et al. [129]. This analysis is, however, conducted as a refinement after the SPARK callgraph is done. It replaces the points-to object in the Soot scene, but has no influence on the callgraph. Therefore, without further considerations (such as iteratively re-computing the callgraph after the refinement step), it does not solve the problem at hand. Alternatively, there is the 1-context-sensitive Paddle algorithm [83] for Soot which is intended as a more precise replacement for SPARK. Since Paddle is maintained as a separate component outside of the Soot source code repository, analysis components are usually not designed to work with Paddle. FlowDroid is based on an IFDS solver which in turn uses an interprocedural control flow graph (ICFG). The architecture of the ICFG and the solver would need to be adapted to allow for context-sensitive queries with Paddle which would be a major undertaking. Furthermore, at least Paddle's memory-efficient version based on BDDs is reported to be unstable for larger analysis targets [144].

An alternative technique called *geometric encoding-based context-sensitive points-to sets* has recently been proposed and implemented into Soot by Xia and Zhang [145]. This technique is reported to be 81.9 times faster than Paddle for medium-sized test cases. There is no comparative data for large test sets, because Paddle did not complete on them. In comparison to SPARK, computing the geometric points-to takes about 118% longer and requires approximately 187% more memory. In summary, even this comparatively efficient implementation of a context-sensitive callgraph comes with a performance and memory penalty. As mentioned above, longer callgraph construction times can be compensated by more precise results when performing a static data flow analysis, because a lower amount of spurious call edges must be processed. We therefore see experimentation with context-sensitiver callgraphs in FlowDroid as an important area of future work, despite the higher cost during callgraph construction. On the technical level, FlowDroid, the ICFG and the solver all were implemented before the work of Xia and Zhang was available. Therefore, even for integrating their approach, significant changes to these components are necessary.

In Section 4.11, we report on how FlowDroid propagates types together with taint abstractions to filter out invalid callgraph edges at least in those cases in which an access path rooted in the base object of a virtual method call is tainted. While this type propagation is less precise than a fully context-sensitive callgraph, it only requires little overhead and can thus serve as a middle ground between context-sensitive and context-insensitive callgraph algorithms when they are used in a data flow analysis.

4.2.7 The Rule Engine

The FlowDroid Core supports many features of the Java programming languages (exceptions, arrays, etc.) and is therefore complex. To ensure code quality and maintainability, the normal structure of IFDS problems with four different flow function types (normal, call, return, call-to-return) is not sufficient as Lerch and Hermann have shown [80]. Therefore, the four IFDS flow function implementations in the FlowDroid Core only handle simple taint propagations like mapping taint abstractions from caller to callee and back or handling assignments that only copy taints as-is. These are the flow functions shown in the previous sections. More involved analysis tasks such as modeling arrays and exceptions have been factored out of the Core's IFDS flow functions and into a *Rule Engine*. The Rule Engine can be seen as a plug-in framework for IFDS flow functions. Every plug-in (or *rule* as it is called in FlowDroid) handles a specific language feature or analysis task. This also makes the analysis configurable by optionally joining in or leaving out some of the rules. At the moment, there are the following rules:

- **Source Propagation Rule** Injects unconditional taints at the positions identified by the *Source Sink Manager*, see Section 4.3.
- **Sink Propagation Rule** Checks whether a sink identified by the *Source Sink Manager* has been reached, see Section 4.3.
- **Array Propagation Rule** Models array constructions, reads, and writes, see Section 4.4.
- **Exception Propagation Rule** Models exception throwing and catching, see Section 4.5.
- **Implicit Propagation Rule** Models implicit information flows, see Section 4.6.
- **Strong Update Propagation Rule** Implements strong updates, i.e., kills overwritten taints, see Section 4.7.
- **Wrapper Propagation Rule** Integrates the *Taint Wrapper* into the IFDS flow functions, see Section 4.9.
- **Typing Propagation Rule** Implements type checks to drop taints with incompatible types, see Section 4.11.

Note that rules are lower-level constructs. They are all part of the FlowDroid *Core* component from Figure 5. More specifically, they are part of the IFDS flow functions and only have been separated as a means of good engineering practice. A rule can make use of one or more of the components from Figure 5 described above. In other words, a *component* is a higher-level construct that encapsulates tasks or decisions independently from the IFDS flow functions. The *Taint Propagation Wrapper* component for instance provides external library models. It, however, does not decide when to query for such models or how to propagate them. This lower-level implementation is done by the *Wrapper Propagation Rule* which knows that library models must be applied in the IFDS call-to-return flow function and that taints generated from library models must afterwards be forwarded to the alias analysis, etc.

Though FlowDroid uses its own implementation of an IFDS solver (see Section 4.12), it inherits the concept of pointwise propagation originally proposed by Naeem et al. [100]. Instead of eagerly creating the full supergraph and then applying the analysis, the supergraph is computed on demand. In other words, FlowDroid's IFDS engine starts with unconditional facts at the sources and then maintains a worklist of facts for which successors still need to be computed to extend the supergraph. These graph extensions, or successor computations, are done independently of each other in unsynchronized threads for performance reasons and to allow for a simple model of computation. Therefore, neither the FlowDroid Core, nor any rule can ever assume to have a full picture of the supergraph.

With this general design decisions, rules can also be computed independently of each other. The Data Flow Engine passes the incoming taint at a statement to the FlowDroid core as well as to the rules. It then computes the union of the results generated by the Core and those generated by the rules to obtain the result of the current IFDS flow function, i.e., the new data flow facts used to extend the supergraph. This implicitly also propagates the incoming taint onward in the normal flow function. Conceptually, rules may, however, also conclude that a certain data flow fact must be erased, for instance in the case of the Strong Update Rule. To allow for such rules, the Data Flow Engine provides two flags to all rules: `killSource` and `killAll`. A rule can set either flag and thereby influence how the rule results are joined in the Core Engine. The `killSource` is used to signal to the Engine that the incoming taint shall not be propagated onward in the default implementation of the normal flow function. The `killAll` immediately aborts all taint propagation for the current incoming taint at the current statement under the current context and returns an empty set of data flow facts to the IFDS solver. Note that once a rule sets this flag, the engine also discards all results provided by other rules. In other words, the *kill* flag is global for all rules. In that case, an empty set of flow facts is returned as the outcome of the IFDS flow function.

The IFDS flow functions can only propagate taint abstractions. For each node in the program's control flow graph, they take an incoming abstraction and create zero or more outgoing ones that are then propagated to the next node in the graph. This approach, however, requires the initial taint abstractions to be injected before starting the propagation. FlowDroid therefore iterates over all statements in all reachable methods of the program and checks whether they are a source. If so, a special taint abstraction called the *zero abstraction* is injected at this statement. It models an unconditional taint that is valid in all contexts. This initial injection is handled by the FlowDroid infrastructure when it initializes the IFDS solver. The zero abstraction is special as it does not reference an actual access path. Therefore, none of the flow functions built into the FlowDroid core is able to process it. These functions immediately discard zero abstractions. Actually handling the zero abstractions has been factored out into a special *Source Propagation Rule* (which is part of FlowDroid's rule engine explained in Section 4.2.7). Whenever it encounters a zero abstraction, it creates a new abstraction for the access path tainted by the source and discards the zero abstraction.

Note that the two steps are not identical. While the framework in its first step of creating zero abstractions only needs to know whether a certain statement can be a source in general, the second step, which actually generates the correct taint abstractions, needs to precisely find the right tainted access paths. A statement can not only be a source, because it calls a source method and saves the unconditionally-tainted return value. It can also, e.g., pass a data object to a source method, which then writes the tainted data into a field of that object. In both cases, the respective statement is a source, but the tainted access paths will be different. A detailed discussion of sources and sinks on a conceptual level can be found in Section 3.2. To technically encapsulate all the possible variants of sources and sinks, FlowDroid provides an interface for a *Source Sink Manager*. With this interface, analysts can defined arbitrary rules of what constitutes a source or a sink. While the source propagation rule is responsible for the low-level integration of sources into the core's IFDS flow functions, the source sink manager resides on a higher level of abstraction. It has the following two functions:

- **getSourceInfo()** This method takes a statement and returns a set of access paths that shall be unconditionally tainted. This set can be empty if the given statement is not a source or contain an arbitrary number of access paths. The set semantics is helpful in case a source taints multiple elements at once, e.g., by writing sensitive data into multiple fields of objects passed in as arguments to the source method.

- **isSink()** Given a statement and an incoming access path, this method decides whether a sink has been called. Note that the access path is important, because methods may consume multiple call arguments, but only leak a subset of them. In such a case, a sink should only be considered as triggered if the "right" parameter is tainted at the call.

For integrating the sink handling into the IFDS flow functions, FlowDroid uses a second rule, called the *Sink Propagation Rule*. For each statement and incoming taint abstraction, it queries the source sink manager registered with the analysis. In addition to providing the technical glue code between the source sink manager and the IFDS flow functions, the two propagation rules (source propagation rule and sink propagation rule) also take care of additional lower-level constructs such as caching, or, in the case of sources, triggering the alias analysis where necessary.

FlowDroid provides a default implementation for Java programs that assumes that sources are always method calls. In general, this need not be the case, and a custom source sink manager can define arbitrary statements as sources or sinks. Our default implementation checks whether the target method is listed in a file called `SourcesAndSinks.txt`. If so, its return value is unconditionally tainted. In this simplistic model, sources never taint any parameters passed to them. Only if the method returns `void` or the user code does not process the return value (i.e., the calling statement is not an assignment), and the callee is not static, the base object of the call is tainted instead. When unconditionally tainting a source value, all fields within it are tainted as well, i.e., the generated access path always ends with an asterisk. While this definition is based on a rule-of-thumb, it has proven to be effective in practice. Usually, sources do not fill data objects, but rather return new ones, and assuming that all values in such objects are sensitive is sufficiently precise in public. Especially in contexts such as Android, a permission check (if a special permission is required) happens when the source method is called. Therefore, whatever is returned by this method is guarded by the check and can rightfully be assumed as tainted. Additionally, these simple semantics match the source and sink definition of SuSi [109] from which we take the default `SourcesAndSinks.txt` file. Note that the source and sink definition from SuSi does not require all defined methods to be concrete. If a method

```
1 void test() {
2   String[] arr = new String[3];
3   arr[0] = ``Hello World'';
4   arr[1] = source();
5   leak(arr[0]); // false positive
6 }
```

Listing 13: Over-Approximation for Arrays

```
1 void test() {
2   String[] arr = new String[3];
3   arr[0] = source();
4   arr[1] = source();
5   arr[3] = source();
6 }
```

Listing 14: Index-Based Tainting for Arrays

in an interface or an abstract method inside an abstract class is marked as a source or sink, this definition applies to all implementors of that method.

In some cases, this simple default implementation is insufficient, though. FlowDroid provides an alternative source sink manager that captures the full expressiveness of the interface and requires a precise source and sink definition in an XML file. In this file, the analyst can mark individual access path rooted in the method's return value, base object, or any of the call parameters as tainted, and can explicitly specify whether to taint fields inside, i.e., add the asterisk to the access path or not. To the best of our knowledge, there is no approach that would automatically generate such a precise source and sink configuration, though. Therefore, it can only be assembled manually and is likely to be incomplete.

For dealing with Android apps, we provide an Android-specific source sink manager that is derived from the default implementation (the one that uses simple rules and only requires method signatures as inputs). In Android, special lifecycle methods and callbacks are invoked by the operating system which we precisely model. The parameters passed to these methods can also carry sensitive information and must therefore be considered as sources as well. Furthermore, user interface controls can also be sources, e.g., in the case of password fields. Consequently, the Android-specific source sink manager must be tightly coupled with the analyses that detect such special sources. More details are given in Section 5, where we describe the Android-specific implementations of the FlowDroid interfaces.

4.4 Array Tracking

FlowDroid supports tracking tainted data in arrays. Note that arrays, unlike the Java collection classes, are handled directly in the FlowDroid Core, because they are language constructs. For collection classes, the data flow tracker relies on the generic handling of library methods through taint wrappers that we explain in Section 4.9. Array elements are referenced using a numeric index which may be computed dynamically and thus be unavailable to a static analysis tool unless sophisticated pre-analyses are employed, e.g., through constraint solving [98]. Therefore, we opted to not distinguish the individual elements of an array and rather conservatively over-approximate the taint status of arrays. In other words, whenever a tainted value is written into an array, FlowDroid assumes the entire array to be tainted. If the code under analysis later reads back data from a different index of the same array, this data is considered tainted as well (because it was read from a tainted array) and a false positive may occur as shown in Listing 13. In practice, we have not found this to be an important problem. Therefore, we leave a more precise array handling to future work.

Arrays not only have elements, but also a `length` field which can be used to store sensitive data. As opposed to elements, the size can only be written once, i.e., when the array is created. Changing the array length afterwards is not possible. Still, the value can be read at every code position where the array is accessible. With the over-tainting technique presented above, creating an array with a length taken from a tainted variable would taint the whole array right from the start, though it does not yet contain a single element[14]. To avoid this problem, FlowDroid can optionally distinguish whether the array as such (*contents*), only the length, or both are tainted. This flag is attached to the access path.

A similar technique could theoretically also be used for tracking statically-available array indices. If data is written into the array at a static index (i.e., a constant index available in the bytecode), this index value is stored in the taint abstraction. When data is read from the array, the result is only tainted if the accessed index is either a statically-available constant that is equal to the index in the access path, or if the accessed index cannot be determined statically and there is at least one tainted index in the array. In the latter case, the analysis would

[14] Note that direct element specification such as `String[] arr = new String[] {“First”, “Second”};` is broken up into an array creation with only the length and two consecutive assignments in the Jimple IR

have to over-approximate and assume that the access to the unknown index references a tainted index of the array. Furthermore, if a write operation happens to an index that cannot be determined statically, the complete array would have to be tainted as usual to conservatively over-approximate the missing information. While this technique would potentially increase the precision of the array handling, it can also lead to an increased number of taint abstractions to be propagated. In the example in Listing 14, all indices of the array are tainted in the end, so the most concise taint representation would be a single access path that references the array as such. When tainting the whole array as soon as a single element is tainted, this is trivially achieved. The same taint gets created again for each element assignment, which does not increase the size of the overall taint set, because all those taints are equal. When tainting individual indices, however, all taints are only reference a single element, and thus are separate and will stay separate. IFDS performs a simple set union as merge operation, so even if, in the end, all indices inside the array are tainted, there is no possibility to merge the three taint abstractions into one. Furthermore, if not all, but only some array indices are tainted, one would need not only an IFDS merge extension, but also a way to efficiently represent index ranges for not increasing the number of taint abstractions to be propagated. As explained above, we leave these issues to future work. Dillig et al. developed an approach to more precisely analyzing the contents of collections and arrays [38]. The required analysis effort is non-trivial, but integrating such an approach into FlowDroid could potentially increase the precision of the analysis.

4.5 Exception Tracking

FlowDroid is able to track data flows over exceptions. This includes two aspects. Firstly, the normal data flow propagation in the IFDS solver must operate on an interprocedural control flow graph that contains exceptional edges. It must take into account that control flow may not only progress in a linear way, but may also jump to an exception handler if the current statement throws an exception. In the example in Listing 15, the statement in line 4 throws an exception and transfers the control flow to line 8 where a leak occurs. Without exceptional edges in the control flow graph, this leak would be missed. Furthermore, in this particular example, the exception is not optional, but is always thrown (unlike, e.g., a field access that throws a `NullPointerException` if and only if the base object is `null`). Consequently, the control flow will always be transferred, line 5 is dead code, and that leak can never occur.

These (additional) exceptional control flow edges are captured by Soot's `Exceptional Unit Graph`. It is based on a `Throw Analysis` that, for each statement, defines that maximum set of exceptions that it may throw. A field access may, for instance, throw a `NullPointerException`. The exceptional unit graph adds an edge for each possible exception to each catch handler that can possibly catch this exception type and that is defined over a range containing the throwing statement. This approach implicitly assumes that every syntactically possible exception is also thrown, i.e., it does not check whether some exceptions are actually impossible. In the case of the field access, it does not prove that there is actually a chance for the base object being null which is a necessary precondition for the exception to be thrown. It therefore provides a conservative over-approximation over the exceptional control flow edges. According to the JVM specification, some exceptions such as the `OutOfMemoryError` can always happen. Therefore, if there is an exception handler for `java.lang.Throwable`, which is the common superclass of all exceptions and errors in Java, there will be an edge from each statement to this handler. Note that the throw analysis is platform-specific. Soot provides one implementation for Java and one for Android. In Android, many general exceptions such as `OutOfMemoryError` or `VirtualMachineError` do not exist. Instead, the operating system directly terminates the app if such an error occurs. For not producing spurious edges, it is therefore important to chose the right throw analysis when constructing the interprocedural control flow graph.

Secondly, the data flow tracker must handle tainted data that is part of an exception object that is thrown. Consider the example in Listing 16. First, sensitive data is obtained from the source. In line 4, an exception with the tainted data as its message is thrown. In line 7, this exception is caught and the data is read from the message, before it is leaked in line 8. For capturing these semantics, the data flow tracker must correctly propagate the taint on the exception object created in line 4 to the exception object caught in line 7, though these might reside in totally different local variables. In FlowDroid, handling such taints inside exceptions is done by the *Exception Handling Rule*. When a new exception object is created and tainted data is written into that object, the data flow core's built-in flow functions are still sufficient for creating the access path referencing the data inside the object. In the example, FlowDroid would create a taint on `e1.message` given that `e1` is the original temporary variabe in which the exception is stored before it is thrown. Once the exception is thrown, however, special handling is required. The exception handling rule sets the *exception thrown* flag. This flagged abstraction is then propagated to the `catch` block by following Soot's normal exceptional control flow graph. When processing the first statement in the `catch` block, the exception handling rule checks whether the incoming taint has the *exception thrown* flag set.

```
1  void test() {
2    String s = source();
3    try {
4      throw new
           RuntimeException(``Hello
           World'');
5      leak(s);
6    }
7    catch (RuntimeException ex) {
8      leak(s);
9    }
10 }
```

Listing 15: Exceptional Control Flow

```
1  void test() {
2    try {
3      String s = source();
4      throw new RuntimeException(s);
5    }
6    catch (RuntimeException ex) {
7      String t = ex.getMessage();
8      leak(t);
9    }
10 }
```

Listing 16: Tainted Data in Exception Object

If so, it is converted into a tainted access path that references the local variable of the caught exception, regardless of the original tainted variable in the incoming access path. In other words, the flag makes sure that the base of the original access path is ignored, and is replaced by the variable that hosts the caught exception when entering the `catch` handler.

Technically, Soot's inter-procedural control flow graph combines the traditional callgraph with an intra-procedural exceptional control flow graph. Consequently, there is only an edge from a statement that throws an exception to the respective exception handler in the same method such as in the examples in Listing 15 and 16. This edge is part of the normal intra-procedural exceptional control flow graph. If the exception escapes from the current method and is caught inside the caller, there is no edge from the throw site to the handler. Nevertheless, the IFDS problem can handle this case as follows. The throw site in the callee is the origin of a return-flow edge that models that the control flow immediate leaves the callee when the exception is thrown. In the case the exception is optional (e.g., only happens when a base object of a field access is null), there is also a normal flow edge to the successor statement that models the case in which the exception is not thrown. In either case, inside the caller, there is an intra-procedural control flow edge from the call site to the exception handler as defined by the may-throw analysis on the call site. In summary, there is no direct edge from the throw site to the exception handler, but rather two consecutive edges: One that leaves the method (intra-procedural inside the callee) and one from call site to handler (intra-procedural inside the caller).

Figure 6 graphically shows the IFDS flow functions for a simple example. Note that in this example, the exception is thrown under some opaque predicate `if(?)` to depict that it is only thrown under cetain circumstances and that there is also a normal control flow onwards. This is important to not make `leak(b)` unreachable in the caller, but show how exceptional edges are handled together with normal ones. Further note that there is no fall-through from `leak(b)` to the exception handler in the caller. The caller returns after `leak(b)`. For the exceptional flow from `callee()` to the handler in `caller()`, you can see that the original abstraction on variable `f` is first transferred back to the caller, even though variable `f` is not even in the scope of the caller. However, because the exception flag is set, the normal flow function that processes the catch statement will map it to the right receiver `e`. The intermediate flow fact is immediately discarded afterwards. These are the two consecutive data flow edges along the two control flow edges ((1) return from `callee()` and (2) into the handler in `caller()` that are necessary to model this flow.

4.6 Implicit Data Flow Tracking

The concepts discussed so far focused on explicit data flows, i.e., sequences of assignments that propagate data through the program. It is always the sensitive data item itself that is passed on, either completely, or in parts (i.e., by taking a substring of a tainted string). Implicit flows, on the other hand, do not refer to the actual data being propagated, but information about the data. In the example in Listing 17, not the string obtained from the source is leaked, but the information whether that string starts with *Hello*. In some cases, this can already violate security or privacy policies. As an extreme example, a malicious program could, for instace, iterate over a string and, for each character in the string, first send out whether it is equal to "a", then send out whether it is equal to "b", etc. This is done for all possible values of all characters in the string. With this information, the attacker could completely reconstruct the string, though the string is never explicitly sent out by the program.

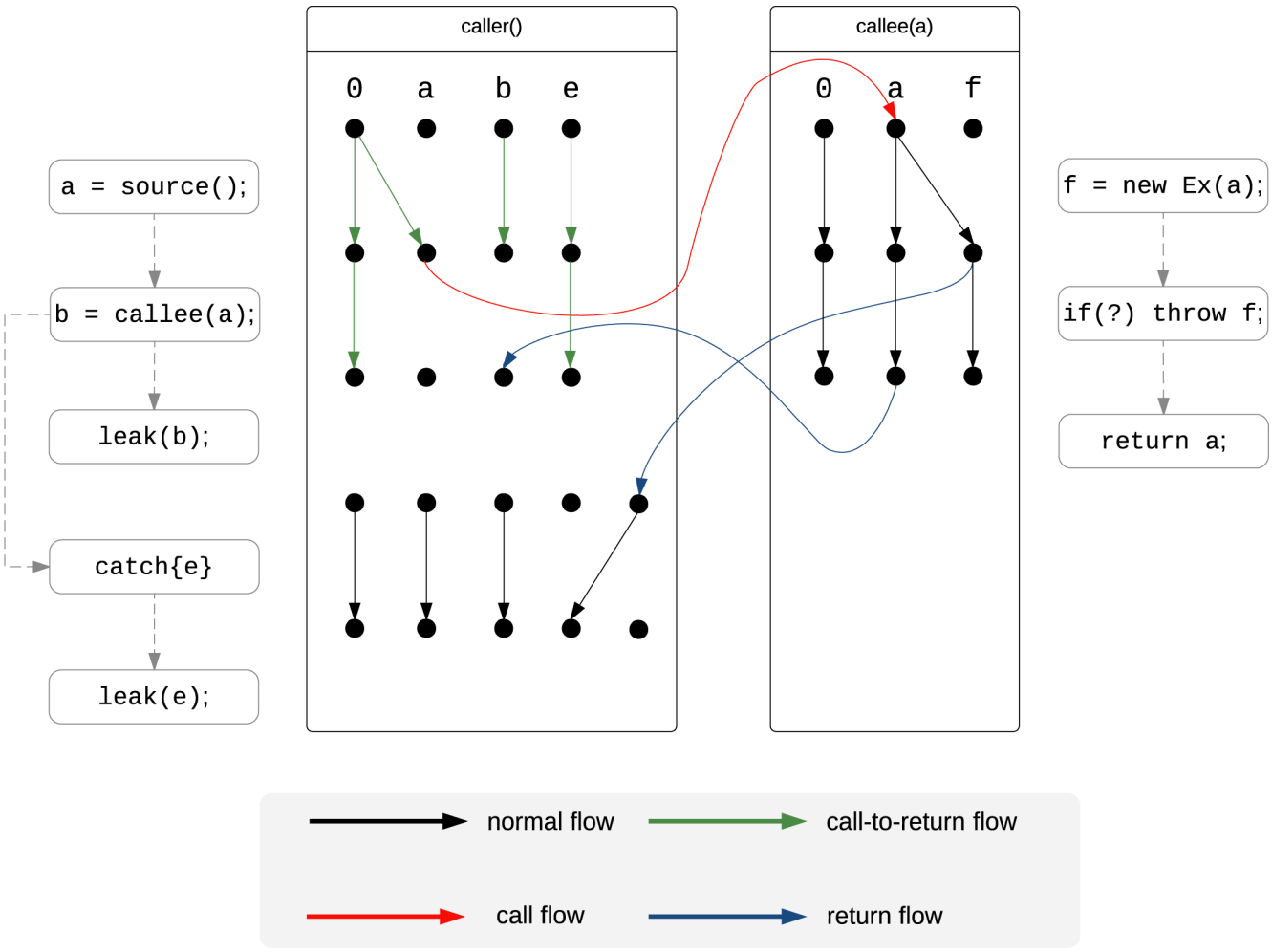


Figure 6: IFDS Flow Functions for Exceptional Data Flow

```
1 void test() {
2   String data = source();
3   boolean b;
4   if (data.startsWith("Hello"))
5     b = true;
6   else
7     b = false;
8   leak(b);
9 }
```

Listing 17: Implicit Flow Example

```
1 void test() {
2   String data = source();
3   boolean b;
4   if (data.startsWith("Hello"))
5     leak("Starts with Hello");
6   else
7     leak("Does not start with
          Hello");
8 }
```

Listing 18: Implicit Flow Example 2

```
1  void test() {
2    String data = source();
3    int i = 0;
4    if (data.startsWith("Hello") {
5      i = 42;
6      leak("Hello World");
7    } else {
8      i = 42;
9      leak("Hello World");
10   }
11   leak(i);
12 }
```

Listing 19: Equal Branches and Implicit Flows

```
1  void test() {
2    String data = source();
3    if (data.startsWith("Hello") {
4      foo();
5    leak(this.a);
6  }
7
8  void foo() {
9    leak("Hello World");
10   this.a = 42;
11 }
```

Listing 20: Interprocedural Implicit Flows

While not tracking implicit flows can lead to leaks being missed which are highly relevant in the context of security and privacy, implicit flow tracking also comes with its own challenges and disadvantages. Conceptually, an implicit data flow tracker taints all data items that depend on any data element that is already tainted. Even in benign programs, this can lead to a large number of additional taints that not only aversely impact performance, but also produce unexpected leak reports. These detected leaks are not technically false positives, but merely model expected behaviour of the program that just happens to fall under the very broad definition of an implicit flow. If a program, for instance, requires a user to log in, it will always leak information about the real password in the back-end database, namely if the password entered by the user during his login attempt is equal to the correct password or not. This single-bit information leak is unavoidable if the login procedure works as expected, because a login procedure, by definition, tells apart correct from incorrect login attempts. Nevertheless, a leak will be reported. This example already shows, that implicit leaks can greatly clutter the output of the data flow tracker. King et al. [76] investigated this issue and found these inherent problems of implicit flow tracking to likely be the root cause why data flow tracking tools that are used in practice usually focus on explicit data flows only. Therefore, implicit data flow tracking is an optional feature in FlowDroid which is disabled by default. FlowDroid's concept of rules (see Section 4.1) allows us to decouple the complete implicit flow implementation from the rest of the system. Only if the implicit flow feature is enabled, the respective rile is registered with the rule engine.

Recall that FlowDroid is based on tracking access paths. Therefore, the implicit data flow tracker must create new tainted access paths for every left side of an assignment if that assignment is control flow-dependent on a conditional that includes a tainted variable. Before discussing our concrete implementation in FlowDroid, it is necessary to understand that in Jimple code, just like in Java bytecode, conditionals are always explicit. While the Java source code language allows shortcut notations such as `a = b ? c : d`, the Java compiler will always turn this into an explicit conditional: `if (b) a = c; else a = d;`. Therefore, whenever there is an implicit data dependency, there is always a control flow diversion through an `if` statement. Note that `while` loops are also represented as `if` conditions and jumps and are not additional language constructs.

When a taint reaches an `if` condition during normal forward propagation, we create a special taint abstraction on access path `*`. This access path does not refer to a specific variable, but to all of them. Semantically, this abstraction mandates that all left sides of assignments at which this special taint abstraction arrives shall be tainted, without requiring any match on the right side. In other words, this access path serves like a fake variable that encodes control-flow dependencies. Aside from this rule, the special abstraction is propagated just like any other taint. It, however, also contains the postdominator of the `if` statement at which it was created. The postdominator is the first unit after the control flow of the *then* and the *else* branch merge again. When the special taint reaches this statement, it is deleted. Successor statements from here are no longer control flow-dependent on the conditional[15]. For finding the postdominator of a given `if` statement, FlowDroid leverages Soot's existing postdominator analysis.

Aside from assignments, calls to sink methods also need special treatment. In the example in Listing 18, only constant strings are passed to the sink method. Nevertheless, an observer who receives this string is able to deduce whether the sensitive data obtained from the source started with "Hello" or not. To capture this kind of leaks, FlowDroid always records a leak when the special taint `*` arrives at a call to a sink method, regardless of the

[15] The control flow always merges at the end of the method at latest. This is the fallback if Soot's postdominator analysis is unable to find a more precise postdominator.

```
1  void test() {
2    int data = source();
3    String[] arr = new String[data];
4    try {
5      arr[42] = "Test";
6    }
7    catch (ArrayIndexOutOfBoundsException
         e) {
8      leak("Hello World");
9    }
10 }
```

Listing 21: Exceptional Control Flow, Implicit Data Flow

parameters passed to the method. While this technique ensures that no implicit leak is missed, it can also lead to false positives. In the example in Listing 19, the call to the sink method is control flow-dependent on a conditional on tainted data. However, regardless of the outcome of the check, the same constant value is leaked and the same state modifications are performed. Consequently, an external observer that only has access to the values passed to the sink methods, cannot deduce anything about the original sensitive value obtained from the source. A static tool such as FlowDroid can only avoid such false positives by proving equality between the two branches. For more complex code with nested conditionals, we would have to prove that all possible control flow paths through the branches are equal. This is a problem FlowDroid does not address. Still, in simple cases such as the one shown in Listing 19, FlowDroid's constant string propagation (see Section 4.14) is able to detect that the value of variable `i` is always `42` in Line 11. Therefore, the call argument is replaced by the constant value `42`, which is no longer syntactically dependent on any secret value, and the false positive is avoided.

All of the examples discussed so far focused on intra-procedural implicit flow tracking. In Listing 20, method `foo()` is only called if the sensitive data starts with "Hello". Conceptually, the very same rules apply if we inline the body of `foo()` into the respective branch of the conditional. To simulate this inlining during the taint analysis, FlowDroid has a special treatment for the * taint at call sites. Normally, this taint is propagated in addition to all other, "normal" taints. This is important, because the other taints are needed again after the postdominator has been reached, so they cannot be discarded on the way. In the case of a method call, however, a context change happens. All of the contents of `foo()` are captured by the * taint that taints everything. Inside `foo()`, there is no postdominator after which the * taint is removed and the normal, selective tainting continues. Therefore, it is sufficient to only propagate the * taint into `foo()`. If there are other taints, for instance on `this.x`, they are irrelevant inside `foo()`, because they are already covered by the much broader * taint. FlowDroid therefore keeps a list of contexts, statements, and methods to denote into which method a taint * has already been propagated at which call site in which context. If such an entry exists and an explicit taint arrives at such a call site with an already-known context, it is not propagated into the callee[16]. Only propagating the * taint into the callee (and all of its subsequent callees) can significantly reduce the time and memory consumption of the analysis. Technically, it is important to note that if we omit taints on the call flow function, we must make sure to propagate them over the call-to-return flow function for not losing them inside the caller. Even if we do not need a taint on, e.g., `this.x` inside `foo()`, it must still be there inside `test()` when the call to `foo()` returns.

Implicit flows can also be triggered through exceptions. In the example in Listing 21, the exception is only thrown if the array has less than 43 entries, i.e., the value obtained from the source was less than 43. If that is the case, a constant string is leaked in the exception handler, allowing an observer to derive information about the sensitive value. To capture such leaks, FlowDroid must use a slightly broader definition of the normal flow function than normally used. The outcome of the normal flow function is not only dependent upon the incoming taint and the current statement, but also upon the next statement at which the control flow will continue. FlowDroid checks whether this control flow edge between current and next statement is an exceptional edge. If this is the case and the current statement references a tainted value, a new * taint is generated. In other words, if a statement uses a tainted variable and then throws an exception, we assume that the fact whether this exception is thrown or not

[16] Technically, this optimization introduces a race condition by design, because the analysis cannot know whether the * taint or the concrete taint on `this.x` will reach the call site first. This can lead to the optimization being applied only partially, but cannot affect correctness.

```
1 void test() {
2   Container c1 = new Container();
3   Container c2 = c1;
4   c1.data = source();
5   c2.data = null;
6   leak(c1.data);
7 }
```

Listing 22: Strong Updates and Aliasing

can potentially depend on the tainted data. The exception handler must then be treated in the same way as a conditional branch.

This approach can, however, also lead to false positives. In the example in Listing 21, the length of the array is potentially lower than 43. Assume that the array initialization were `String[] arr = new String[data + 100];` instead. The array size would then always be larger than 43, and no exception would be thrown. Consequently, there would not be any leak. FlowDroid would, however, still report a leak if implicit flow tracking is enabled. Recall that in FlowDroid, the program's control flow is modeled through an interprocedural control flow gaph (ICFG). For exceptional flow edges, this graph is based on Soot's general-purpose `ExceptionalUnitGraph`, which, in turn, is based on an interprocedural may-throw analysis. This means that there is an exceptional edge from a statement `stmt` to a handle `hnd` in the ICFG if `stmt` can potentially throw an exception that is caught by `hnd`. An array access can potentially always throw an `ArrayIndexOutOfBoundsException` and thus, there will always be a control flow edge to the respective exception handler if one exists. Soot's may-throw analysis does not perform range checking. This is yet another example of additional leaks that are reported when implicit flow tracking is enabled.

4.7 Strong Updates

The normal data flow rules built into FlowDroid's data flow core only capture generating new taints. In some cases, however, existing taint abstractions must also be removed, i.e., not propagated onward. This happens, for example, when a tainted variable is overwritten with non-tainted data. In the simple case, the left side of the assignment exactly matches a prefix of the incoming access path. This simplistic view, however, fails on aliases. In FlowDroid, aliases are first-class taints as explained in Section 4.8. Whenever a heap object is tainted, FlowDroid enumerates all aliases for this object if the default aliasing strategy is used. When an alias is found for a specific taint, it becomes an additional taint on its own. Consider the example in Listing 22. On line 5, there are two incoming taints: `c1.data` and `c2.data`. Therefore, killing the original taint `c1.data`, because the left side of the assignment matches that one access path, leaves the taint modeling the alias (`C2.data`) intact. In the example, this leads to a false positive in line 6. Recall that taint propagation is point-wise in IFDS, so a flow function is always computed on a single incoming taint. When killing the taint on `c1.data`, there is no possibility to influence what happens to the taint on `c2.data` on that line.

To avoid such false positives, FlowDroid provides the *Strong Update Propagation Rule*. For each incoming taint, if it *must alias* with the left side of an assignment, this taint may not be propagated onward. In the example, the taint on `c2.data` is trivially killed on line 5. The base object of the access path `c1.data` must-aliases with the base object of the referenced field `c2.data`, so this taint is killed as well. Note that this requires a must-alias analysis, while the normal taint propagation that generates new derived taints, is based on a may-alias analysis. FlowDroid therefore relies on Soot's built-in intra-procedural must-alias analysis for such queries. As a consequence, if aliases are created in other methods, the analysis will conservatively degenerate and keep the alias taint on the alias alive even if the original taint is killed.

We perform special-casing for the dummy main method in which, by construction, no must-alias relationships exist. Skipping this method improves the analysis performance, because the dummy main method can reach a significant size with a large number of local variables. The more local variables there are, the more potential relationships must be checked, which is avoidable in this case.

4.8 Precise and Efficient Alias Analysis

A precise static data flow analysis tool must also precisely handle aliasing relationships. In the example in Listing 23, the tool must capture that not only `a.fld`, but also `b.fld` is tainted when the sink method is called at line 13. There are two possible models to capture such aliasing relationships during the static data flow analysis:

- **Eager tainting** Whenever a heap object is tainted such as `a.fld` in line 12, also enumerate and taint all aliases. From this line, the analysis would therefore propagate two taints onward: `a.fld`, and `b.fld`. When the analysis reaches the call to the sink in line 13, the leak can be detected by simple pattern matching against the incoming taints.

- **Lazy tainting** When processing an assignment, only the receiver access path is tainted, aliases are ignored. In line 12, the analysis would only propagate `a.fld` onward. Whenever a heap object is read (line 13), the analysis must then, however, check for aliases to discover that the leaked access path `b.fld` aliases with the incoming taint `a.fld`.

```
1  void bar() {
2    // All of the relevant behavior
3    // is in the callee
4    foo();
5  }
6
7
8
9  void foo() {
10   A a = new A();
11   A b = a;
12   a.fld = source();
13   leak(b.fld);
14 }
```

Listing 23: Simple Aliasing Example

```
1  void bar() {
2    X x1 = new X();
3    X x2 = new X();
4    X x3 = foo(x1, x2);
5    leak(x3.a.fld);
6  }
7
8  X foo(X p1, X p2) {
9    A a = new A();
10   X x = new X();
11   x.y = a;
12   a.fld = source();
13   return x;
14 }
```

Listing 24: Complex Aliasing Example

In the literature on alias analysis, the aliasing problem is often formulated as a binary decision: Given two variables, can they potentially alias? When integrating such a binary alias analysis into a static data flow analysis tool, lazy tainting is the natural choice. Whenever the solver processes an incoming tainted access path and a current statement, it can check whether the base variable of the incoming taint may alias with any variable that is used in the current statement. The question whether an incoming taint can possibly alias with a leaked access path is exactly such a binary question. For implementing an eager tainting strategy based on binary decisions, one would have to iterate over all statements in the code as soon as a heap object is tainted, and check whether this new taint may alias with a variable used in any statement. If so, it would then need to create a taint for that variable at the respective statement as well. This requires one loop over all of the program's code for each newly tainted heap object, which can easily lead to significant overhead. In contrast, the lazy tainting strategy can defer this checking to the point at which the respective statement is being processed by the normal taint propagation anyway. Note that this is not an argument against eager tainting, but rather shows that the classical formulation of the aliasing problem as a binary decision problem is not a good fit for eager tainting strategies. Instead, one would need the alias analysis to enumerate all aliasing access paths efficiently without requiring additional passes over the program code for each newly-tainted heap object. We will discuss this issue in more detail later in this section.

While efficient eager tainting requires changes to the interface of the alias analysis (and in fact an extension to what alias analyses provide to their clients), lazy tainting also has its drawbacks. In simple examples such as the one from Listing 23, eager tainting and lazy tainting seem equivalent. In general, lazy tainting can, however, quickly become inefficient. Consider the more complex example in Listing 24. When the analysis returns from method `foo`, it needs to map back the taints into the context of the caller `bar`. Those taints that reference objects which are not visible in the caller are dropped. If eager tainting is used, this process is straightforward: At line 13, the analysis processes two incoming taints: `x.y.fld` and `a.fld`. The callee-side access path `a.fld` has no corresponding access path in the caller as it is not based on an interface element such as a parameter, this-local, or return value. It is therefore dropped. The taint `x.y.fld` is based on the variable `x` which is returned. Therefore, it corresponds to

`x3.y.fld` in the caller, because the return value of `foo` is assigned to `x3` in the caller `bar` in line 4. In other words, there is only a single taint abstration that needs to be propagated onward inside the caller.

If we analyze the same example with a lazy tainting strategy, the only incoming taint at line 4 is `a.fld`. This access path does not match any access path inside the caller as the callee-side variable `a` is not visible in the caller. Dropping this taint would, however, be unsound as it would also kill the alias which *is* visible in the caller. Remember that taints in lazy tainting reference not only their own access path, but also all possible aliases. These aliases are only made explicit on heap accesses. When returning from method `foo`, the analysis therefore cannot know the exact set of variables that is represented by the taint on `a.fld`. It cannot decide whether, at some point later in the code, an alias of `a.fld` is still accessed or not. Therefore, it cannot decide whether this taint needs to be mapped back into the caller or not. The analysis is supposed reason about mapping access paths between contexts without having full information yet. Note that though we demonstrate the problem with method returns, this is a generic problem with context changes and lazy tainting that arises on method calls as well. There are two possible methods to implement context changes with lazy tainting:

- **Propagate everything** Propagate all taints onward as-is. In the example, the taint `a.fld` would be propagated onward in the caller `bar` although it does not point to any valid object there. Still, whenever a read operation on a heap object happens, the data flow analysis can use this taint in a binary alias query to check whether it may alias with that heap object. In line 5, the analysis would process the incoming taint `a.fld` and would check whether this access path aliases with `x3.a.fld`.

- **Manifest aliases on context change** When propagating taints over a context change (method call or return), enumerate all aliases of a given tainted access path. Then, map these access paths into the other context. This essentially is a mix between eager and lazy tainting: It applies the eager tainting technique solely to context changes and keeps the lazy technique everywhere else. In line 13, the analysis would take the incoming taint `a.fld` and compute its alias `x.y.fld` which then gets mapped back into the caller method `bar` as `x3.a.fld`.

Note that the propagate-everything technique assumes that, in the analysis, variables are referenced through unique identifiers and not through names. A variable `a` in `foo` must be distinguishable from a variable with the same name `a` in `bar` for not creating false positives when propagating taints onward as-is. It must still be clear that this taint, albeit being propagated in `bar`, references a heap object with a base variable from `foo`. In the Soot framework, on which FlowDroid is based, this is trivially the case, because all variables are modeled using objects in the AST of the Jimple intermediate representation. A variable `a` in `foo` would be a different object in the analysis than a variable `a` in `bar`.

While the propagate-everything technique is sound and does not miss any leaks, it is inefficient, because it cannot prune taints during context changes. It would need to propagate the taint `a.fld` onward even if it does not alias with anything that is visible outside of the scope of method `foo`. Because aliases are only checked when accessing heap objects, there is never enough information to know that a certain taint becomes inccessible and can safely be deleted. Consequently, propagation everything accumulates unnecessary taints and thus negatively affects the scalability of the data flow analysis. This is a general issue with lazy tainting for aliases. Therefore, the second technique (manifesting aliases on context change) compromises on lazy tainting; it is essentially a combination of eager and lazy tainting. Instead of enumerating all aliases whenever a heap object is tainted, it only enumerates aliases on context changes. This enables the data flow analysis to erase those access paths that are not visible in the other context. In the example, the taint `a.fld` would not be propagated back into `bar`.

The combined technique (just like eager tainting), however, requires enumerating all aliases of a given access path. This is not commonly a feature of off-the-shelf alias analyses as Xiao [146] points out (he calls this enumeration function `ListAliases` as opposed to the binary `IsAlias` query). Therefore, most alias analysis are not suitable for the integration into a static data flow analysis tool that uses eager tainting or the combined technique. Despite the great existing amount of research on alias analysis, this area needs to remain under active investigation. This lack of existing algorithms and well-tested implementations inside static analysis frameworks such as Soot is the primary reason for FlowDroid to provide its own alias analysis. One noteworthy exception is Boomerang by Späth et al. which allows exactly such alias enumeration. It was created as a generalization of FlowDroid's flow-sensitive alias analysis which we present in Section 4.8.4 and builds upon the work presented in this thesis.

Before discussing the details of FlowDroid's alias analysis, we will first address why enumerating aliases cannot easily be emulated with a set of binary alias queries. Again, take the example from Listing 24. When returning from method `foo` in line 13, the analysis must know the aliases of taint `a.fld` that are visible through the interface of the method `foo`. Trivially, one would replace the base variable of this taint with interface variables and run binary alias checks:

- Does `a.fld` alias with `x.fld`?
- Does `a.fld` alias with `p1.fld`?
- Does `a.fld` alias with `p2.fld`?

These queries, however, fail to capture the necessary alias information from the example, i.e., the fact that `a.fld` aliases with `x.y.fld`. To capture this alias, one would need to query the binary alias analysis for all possible aliases of `a`. One would need to enumerate all sequences of fields that are accessed inside method `foo` and generate all possible combinations thereof. This would lead to the following additional queries:

- Does `a.fld` alias with `x.y.fld`?
- Does `a.fld` alias with `p1.y.fld`?
- Does `a.fld` alias with `p2.y.fld`?

If the method `foo` would also access other fields such as `x.z`, the data flow analysis would also need to query the alias analysis for these fields. If n is the number of fields accessed in a given method, and the access path length is restricted to l elements, the number of possible access paths is $O(n^l)$. From this example, we can conclude that binary alias queries, at least on a conceptual level, do not effectively capture the requirements of a static data flow analysis.

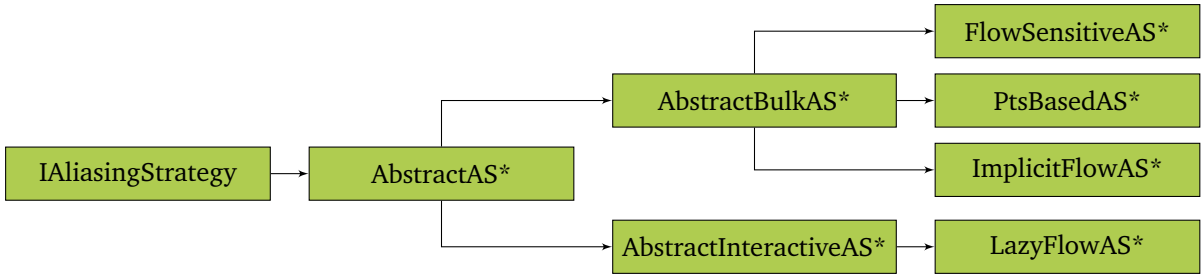
The challenges that arise for the alias analysis are tightly coupled with the choice of the underlying taint abstraction, more precisely: the heap model on which the analysis operates. As explained in Section 3.3, FlowDroid is based on access paths, which is a storeless abstraction according to the terminology of Kanvar and Khedker [72]. If we had instead chosen a store-based abstraction, the analysis could taint abstract representations of the respective heap objects instead of ways to access them (i.e., access paths). In that case, one would only need a single taint abstraction for the heap object, regardless of the number of different variables and sequences of field dereferences that point to the object. This would also eliminate the question of eager vs. lazy tainting, because all these "secondary" taints would no longer be necessary. On the other hand, efficiently summarizing such a heap model without significantly losing precision is a challenge on its own. Merging different memory objects into one abstract heap object leads to false aliases, while providing separate heap objects for what is actually the same memory object leads to missed aliases. Common techniques such as mapping all objects created at the same allocation site to the same abstract heap object, for instance, cannot properly model factory methods.

In the remainder of this Section, we will first discuss FlowDroid's plug-in architecture for alias analyses in Section 4.8.1. We will then show the practical impact of the emulating access-path aliased analysis with an off-the-shelf binary alias analysis in Section 4.8.2. Afterward, we will introduce FlowDroid's custom alias flow-sensitive analysis that is explicitly tailored at computing aliases for access path-based taint analysis problems in Section 4.8.4. In Section 4.8.5, we introduce a special alias analysis tailored at implicit flow tracking. Lastly, we present related work on alias analysis in Section 4.8.6.

4.8.1 FlowDroid's Aliasing Architecture

To support experimentation with different alias analyses, FlowDroid decouples the aliasing problem from the data flow engine through an interface `IAliasingStrategy` as explained in Section 4.1. This interface caters for both eager and lazy tainting. Eager tainting is implicitly asynchronous: Whenever a new heap object is tainted (i.e., a tainted value is assigned to a field or an alias element), the analysis is invoked with a call to `computeAliasTaints`. It is tasked with finding all aliases of the new heap taint, with creating taint abstractions for these aliases, and with injecting these alias taints into the data flow solver. To allow for the *manifest aliases on context change* strategy (the combination that performs eager tainting on method returns, but lazy tainting everywhere else), the alias analysis is also invoked on method returns. To capture the shared semantics of eager tainting and the combination approach, FlowDroid provides an abstract base class called `AbstractBulkAliasStrategy`. For lazy tainting, the aliasing strategy is invoked for a binary may-alias decision on every access to a heap object such as a field or an array index by calling `mayAlias`. For this variant, the abstract base class `AbstractInteractiveAliasStrategy` is provided. Figure 7 shows the complete aliasing class diagram.

Note that FlowDroid also requires an interprocedural must-alias analysis to support strong updates. The `IAliasingStrategy` interface, however, does not provide an abstraction for this aspect. Instead, FlowDroid always relies on Soot's default implementation. We have not seen good reasons for exploring other alternatives for must-alias relationships so far.



*Class name abbreviated. AS = AliasStrategy

Figure 7: FlowDroid's Aliasing Architecture

4.8.2 FlowDroid's PtS-Based Aliasing Analysis

In theory, the conceptual mismatch between access paths of arbitrary lengths and standard off-the-shelf binary alias analyses can lead to a high performance penalty. The number of queries to be made can be high depending on the maximum access path length. To assess the impact in practice, we have implemented such a strategy in FlowDroid. As the underlying aliasing algorithm, we use the `PointsTo` implementation from SPARK.

```
1 void test() {
2   A a1 = ...;
3   A a2 = ...;
4
5   a2.b.c = a1.b.c;
6   a1.b.c.d = source();
7
8   leak(a2.b.c.d);
9 }
```

Listing 25: PtS-Based Aliasing (Java Code)

```
1  void test() {
2    A a1 = ...;
3    A a2 = ...;
4
5    B b1 = a1.b;
6    C c1 = b1.c;
7
8    B b2 = a2.b;
9    b2.c = c1;
10
11   D tmp = source();
12   c1.d = tmp;
13
14   C c2 = b2.c;
15   D d2 = c2.d;
16   leak(d2);
17 }
```

Listing 26: PtS-Based Aliasing (Jimple Code)

Our implementation is based on the observation that field accesses can only refer to the base object and one field in Jimple, e.g., `a.fld`. Longer sequences of field dereferences are always broken up into chains of simple one-field accesses. Listing 25 shows the original Java code of an example. The corresponding Jimple code is shown in Listing 26. The Jimple code has been simplified slightly to focus on the important aspects of the example. In the original Java code, the alias analysis would have to detect the aliasing relationship between `a1.b.c` and `a2.b.c` to find the leak in Line 8. In the Jimple code, the analysis must only find the aliasing relationship between `c2.d` and `c1.d` which only involves the base object and one field instead of the much longer original access path. In other words, the problem of finding aliases on longer access paths is reduced to finding aliases between interim variables that store prefixes of the aliasing access paths. This is a direct consequence of the prefix finding problem discussed in Section 4.8 above. It is only made implicit by the Jimple language construction.

In the following, we will explain the alias analysis process in detail for the given example. The first unconditional taint is created at the source in Line 11. Directly in the next line 12, the tainted data is assigned to a heap variable and the alias analysis is triggered for `c1.d`. It scans over the whole method body to find assignments of `c1.d`, more precisely: assignments of anything that has a points-to set with a non-empty intersection with the points-to set of `c1.d`. Note that this analysis is, by definition, flow-insensitive. Line 9 references `c1` which trivially PtS-aliases with itself (the base of `c1.d`), so a new taint `b2.c.d` is created and injected into the forward solver. This alias then becomes a new taint in its own right and is propagated onward like any other taint. The alias analysis then proceeds to find more aliases, now with the set `{c1.d, b2.c.d}`. In Line 9, the alias `b1.c.d` is added for `c1.d`. This taint is also injected into the forward taint propagation and becomes a taint in its own right. The alias analysis now has a worklist containing `{c1.d, b2.c.d, b1.c.d}`. This search continues until the worklist has reached a fixed point, i.e., no new aliases can be found. Regardless of the continuing alias search, the taint analysis can continue and find the leak as soon as the alias taint on `b2.c.d` has been found and is propagated over the last three statements. The alias analysis always only needs to check whether the base object and, in the case of a field reference, the first field

Algorithm 1 PtS-Based Alias Algorithm in FlowDroid

Require: Method body b, tainted access path ap
Ensure: Injection of alias taints into forward taint tracker

```
1:  Q ← [ ap ]{Initial queue of aliasing access paths}
2:  while Q ≠ empty do
3:    curAP ← Q.pop()
4:    for each Unit u ∈ b do
5:      if isInstanceInvokeStatement(u) then
6:        if curAP.hasPrefix(u.base) then
7:          injectIntoForwardSolver(append(u.base, curAP.suffix), u)
8:      else if isInvokeStatement(u) then
9:        for each Parameter p ∈ u.getParams() do
10:         if curAP.hasPrefix(p) then
11:           injectIntoForwardSolver(append(p, curAP.suffix), u)
12:     else if isAssignment(u) then
13:       if isHeapObject(u.rightOp) ∧ hasNonEmptyIntersetion(pts(u.rightOp), pts(curAP)) then
14:         Q ← append(u.getLeftOp, curAP.suffix)
15:       if isHeapObject(u.leftOp ∧ hasNonEmptyIntersetion(pts(u.leftOp), pts(curAP)) then
16:         injectIntoForwardSolver(append(u.getLeftOp, curAP.suffix), u)
```

are a prefix of an access path in the worklist. The remaining fields in the access path do not need to be checked, they are simply copied over to the new alias taint if the prefix matches.

The PtS-based alias analysis only iterates over the statements inside the current method to check for potential aliases. Note that SPARK is inter-procedural, i.e., each single binary alias query takes aliases created in other methods into account as well. Still, the analysis must make sure to issue the right queries and also ask for potential aliases in those other methods, or it will only discover those aliases that are in scope in the current method. Recall that the alias analysis is not only invoked when a new taint on a heap object is created, but also after a method return has been processed. This allows the alias analysis to pick up the taints that were matched back into the callee by the normal taint propagation algorithm and perform the same intra-procedural scan in the callee. If this again yields new taints on aliases, they are again mapped back as normal taints and again, the alias analysis is triggered for a new scan in the callee's callee, etc. With this technique, the alias analysis does not need to process any context switches on returns on its own, but can completely rely on the IFDS-based taint analysis for this task. The only case that needs special treatment is a method call. The alias analysis must also check whether a parameter or a base object of a method call aliases with an existing taint and then explicitly inject a taint for the concrete variable of that base object or parameter into the taint analysis. The taint analysis will then take care of mapping this new alias taint into the callee. In general, aliases are manifested where they are used, be it in an assignment or in a call site, at the latest, however, when the current method returns. Scanning only inside the current method and injecting the found aliases back into the normal forward taint propagation has the advantage of being able to re-use the context handling of the IFDS-based taint analyzer. This is import, because the alias analysis must be able to inject taints into the taint propagation, which in turn, requires a context. One simple solution would be to always use the `null` context (element is unconditionally tainted). However, continuing the complete taint propagation in a `null` context could lead to a precision loss even in cases that are not directly affected by the points-to analysis.

Algorithm 1 shows the PtS-based aliasing algorithm in detail. It is called for a method body `b` and an incoming access path `ap`. This incoming access path is the one that was originally tainted though the assignment to the heap object for which the alias analysis was invoked. The algorithm takes this access path as the first element of its worklist `Q` in line 1. As long as the worklist is not empty, the algorithm iterates over all statements in the current method. We will first discuss how assignments are handled (line 12). If the current statement is of the form `a = b.c` and the current access path `curAP` is `e.f.g`, the algorithm must check whether the points-to sets of `b.c` (the assignment's right side) and `e.f` (the prefix of the current access path) have a non-empty intersection as shown in line 13. If this is the case, a new alias has been found which must be added to the taint set. In line 14, the access path of the newly-found alias is computed by taking the right side of the assignment (`b.c` in the example) and appending the remaining fields of the access path that are not covered by the field access (`g` in the example). In the example, the new alias `b.c.g` is added to the worklist. It is not immediately passed to the forward taint analysis, though. Aliases are only manifested when they are actually accessed in this alias algorithm. The check whether an

alias is accessed is done in line 15. If the left side of the assignment aliases with the prefix of the current access path (based on a non-empty intersection of the respective points-to sets as explained above), the forward taint analysis is triggered. Similar to the case before in which a new alias was discovered, the algorithm adds the remaining fields to the access path. It then injects this access path into the forward taint propagation as a new taint.

The second possibility for referencing a tainted access path aside from assignments is to use the base object of a tainted access path in a method invocation, be it as the base object of a virtual method call (line 5 in the algorithm) of as a parameter (line 8 in the algorithm). In both cases, the alias algorithm derives a new access path with the respective base object from the method call and the field list from the incoming access path which it passes to the forward taint analysis (lines 7 and 11). The forward taint analysis then takes care of mapping these access paths into the context of the callee where the alias analysis is triggered again if necessary (i.e., when the taint analysis has reached another assignment to a heap object in the callee).

4.8.3 FlowDroid's Lazy Aliasing Analysis

Classic alias analysis not only has the issue of not being able to process access paths of arbitrary length, but it also cannot enumerate all aliases for a given tainted heap object. The analysis user (in our case the taint analysis) must then provide for a means to simulate this missing feature. Our PtS-based alias analysis described in Section 4.8.2 presents such a simulation. To cope with context switches, it manifests the aliases created inside a method when returning from that method or when calling another method. This, however, requires scanning the code and for each statement, checking whether it defines a new alias. Such an approach leads to a high number of alias queries, and is not the originally-intended use of the alias algorithm. As an alternative model, we also implement a true lazy alias strategy that never manifests any alias taints before the point at which the respective heap object is actually accessed. In other words, when a new heap object is tainted, this alias strategy does nothing. When returning from a method call, it does nothing. When a heap object is accessed, it checks whether the incoming taint aliases with the current heap access. For this to be possible, all taints must be propagated into all methods, even if the respective base object is not in scope in the concrete callee at hand. Recall that the lazy alias strategy has no information on whether the callee (or any of its transitive callees) may access a heap object that aliases with this incoming taint. It must therefore conservatively assume that every incoming taint might be accessed through an alias and must retain it.

Consider the example from Listing 27. The taint on `a.fld` is not in scope in method `callee`. More precisely: There is no corresponding access path in the callee, so in the traditional concept of mapping access paths into the callee's context when encountering a call sites, this taint would be discarded. However, if this taint is discarded, there is nothing left to propagate into method `callee`, and the alias analysis inside `callee` would not be able to discover that the field accessed in line 9 actually does alias with tainted data. Therefore, this leak could not be found. We solve this issue by retaining all taints that cannot be mapped into a callee as-is when using the lazy alias strategy. In the example, the taint analysis would propagate the taint on `a.fld` into `callee`, thereby ignoring that the base variable of this access path is not in scope inside the callee. When the lazy alias analysis computes points-to sets, it is irrelevant whether the variable is in scope or not (in fact there is no notion of scope for points-to sets in Soot). Therefore, it can find out that `a.fld` and `x.fld` are aliases for the same runtime object in line 9 and can correctly report a leak.

```
1  void main() {
2    A a = new A();
3    A b = a;
4    a.fld = source();
5    callee(b);
6  }
7
8  void callee(A x) {
9    leak(x.fld);
10 }
```

Listing 27: Lazy Alias Analysis Propagate Everywhere

This strategy of propagating all access paths into all scopes can obviously greatly increase the number of taints that needs to be propagated through the program under analysis. Note that the traditional mapping leads to a large reduction of taints. Methods may have many local variables and many tainted access paths in their scope, of which

```
1  void main() {
2    Data p = new ..., p2 = new ...
3    taintIt(source(), p);
4    leak(p.f);
5
6    taintIt("public", p2);
7    leak(p2.f);
8  }
9
10 void taintIt(String in, Data out) {
11   x = out;
12   x.f = in;
13   leak(out.f);
14 }
```

Listing 28: Source Code for Aliasing Example

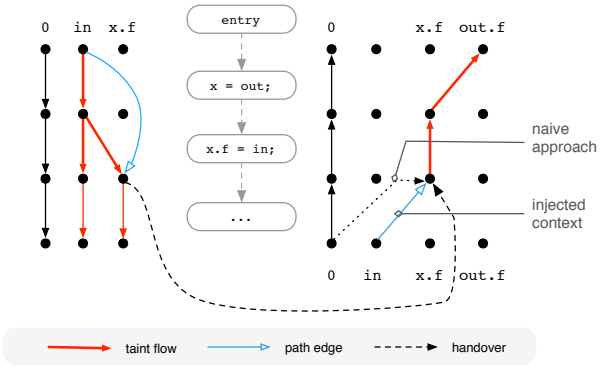


Figure 8: Taint Graph for Aliasing Example

only a fraction gets passed to callees. With the lazy alias analysis, however, all of them must be propagated to all callees and all of their callees, etc. all along the whole call tree. Consequently, this strategy is, while technically sound, infeasible for any realistic program. It merely saves as a baseline for assessing the impact of how bad the impact of traditional alias analysis can be on static data flow tracking. This strategy is not used in any productive settings in which FlowDroid is applied. In our experiments, we find that the analysis runs into timeouts even for moderately-sized apps. Consequently, applying the lazy aliasing strategy to large real-world apps such as those we use for our performance evaluation in Section 8 is infeasible [17].

4.8.4 FlowDroid's Flow-Sensitive Alias Analysis

To achieve full context sensitivity and alleviate the problems with traditional off-the-shelf alias analyses described above, we implemented an alias analysis as an IFDS problem. Basing the alias analysis on IFDS allows it to easily share contexts with the taint propagation which is also implemented as an IFDS problem within the exact same technical framework. Whenever a heap object is tainted during the forward taint propagation, a backward scan for aliases is triggered in a second solver. This second solver works on a reversed copy of the interprocedural control flow graph. If an alias is found during the backward propagation, it is injected into the normal forward propagation as a new taint. In other words, the alias analysis interacts with the taint propagation by injecting flow edges back and forth. While this is a very low-level interface, it means that the forward taint propagation need not wait for alias queries to be answered. It injects the heap taint into the backward solver and continues with its propagation, completely ignoring any aliases. The backward propagation then runs asynchronously. Each time it has found an alias, it injects back an edge into the forward solver which can then propagate the alias as a new, independent taint. This approach exploits the distributivity of IFDS: Instead of computing aliases where they are created, they are computed independently and joined in later as first-class taints. Not having to wait for the alias analysis during taint propagation enables FlowDroid to compute more flow functions in parallel which is especially helpful on modern many-core processors. Note that flow functions are parallelized through worker threads, each evaluation of a flow function is a separate task item that can run on an arbitrary CPU core with free capacity.

Consider the example in Listing 28. In Line 3, the method `taintIt()` is called with a tainted first argument and the object `p` as the second argument which becomes `out` in the callee `taintIt()`. This method then writes the tainted data into `out.f`, and calls the sink method on `out.f`. This immediately leads to a leak. When returning to the caller, `p.f` is tainted as well, leading to the next leak in Line 4. In Line 6, the same method `taintIt()` is called again, but with non-tainted data and a different target object this time. A context-sensitive alias analysis like ours is able to distinguish these two calling contexts, avoiding a false positive in Line 7. Figure 8[18] shows the IFDS flow function evaluations for the `taintIt()` method. The red lines on the left side show the normal forward taint propagation. Initially, the `in` parameter is tainted and this taint is copied over to `x.f`. As this is an assignment to a heap variable, an edge is injected into the backward alias analysis as shown with the dotted black line. Note the change of direction, now the arrows run upward instead of downward on the right side of the figure. These edges

[17] Because all analyses either timed out or ran out of memory we do not report detailed data here.
[18] The figure is taken from our 2014 ICSE paper [9].

```
1 void main() {
2   Data p = new ...;
3
4   p2 = p;
5   sink(p2.f);
6
7   p.f = source();
8   sink(p2.f);
9 }
```

Listing 29: Flow-Sensitivity in Alias Analysis: Source

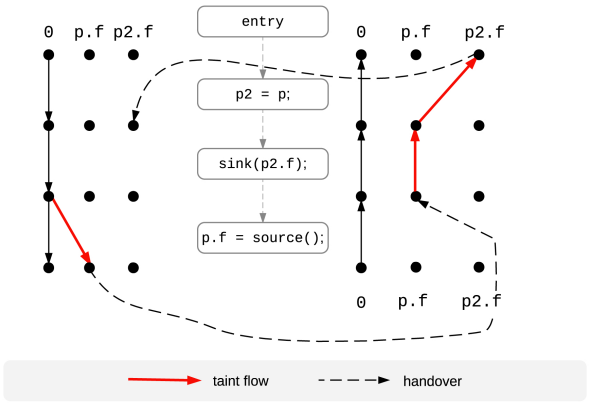


Figure 9: Flow-Sensitivity in Alias Analysis: Taint Graph

lead to `out.f` (red arrows on the right side), discovering the alias. This alias is then injected back into the forward analysis as a new taint.

Recall that IFDS does not store the individual edges between taint abstractions, but constructs and incrementally extends path edges. Instead of storing "`out.f` is tainted if `x.f` was tainted in the previous line", it stores "`out.f` was tainted if the current method was called with `in` tainted". A path edge always connects a flow fact at the start of the method with a flow fact at the current position. During normal forward taint propagation, this is trivial. The forward solver entered method `taintIt` under a specific context (e.g., `in` tainted) and extends the edge from there. This is shown by the blue arrow on the left side of Figure 8. When injecting an edge into the backward solver, it is important to also inject this context into the second solver. Only then, the backward solver can also extend the right path edge, making the discovered aliases also conditional on the context in which `in` was tainted. This is shown with the blue arrow on the right side. In the backward solver, this context originally never existed, because this solver never entered the `taintIt()` method. It, therefore, continues with a foreign context it takes as-is. When a new alias is discovered, it can inject this newly-discovered taint back into the forward solver using this very context to keep context-sensitivity. In other words, whenever a solver (forward taint solver or backward alias solver) enters a method, it must inject this context into the other solver as well. This guarantees that when a solver passes a taint to its peer, this taint is always associated with correct context [19]. A trivial alternate solution that would not inject contexts, but rather link injected edges with the `null` context as shown with the dotted black line on the right side would cause a false positive in Line 7, because it would not have any context in which to inject back the alias on `out.f`. Consequently, this alias would be injected back with the `null` context as well, making the value of the second method parameter unconditionally tainted for all call sites, which would be an imprecise over-approximation.

After tainting `out.f`, the backward alias analysis must return into the caller and look for further aliases in `main()`. For this return edge, it also exploits the injected contexts. The forward solver knows which caller-side contexts lead to which callee-side contexts at which call sites. This information is shared with the backward solver so that the latter can reconstruct the correct caller-side context when returning from the callee. It then continues the alias analysis backward just as in the previous example. Once it reaches a location in which the variable to which the current taint abstraction refers is overwritten, the analysis stops. On the current path, no aliases can be created before that statement. Note that no further actions must be taken, because the discovered aliases have already been injected into the forward taint analysis directly when they were discovered.

Flow-Sensitivity

The IFDS analysis is by design flow-sensitive. The forward taint propagation always associates taints with contexts and the statements at which the taint is valid. The same happens within the backward alias analysis. Retaining

[19] We would like to thank John Toman from the Programming Languages research group at the University of Maryland for identifying and fixing a corner case in the context injection as it was implemented in the FlowDroid open-source project. IFDS uses method summaries. When a solver already has a summary for a method which the other solver then leaves in a new context, the first solver must apply its summary to this new context, even if (due to the summary) no actual propagation needs to be done. In other words, one must be careful with IFDS summaries when injecting contexts.

flow-sensitivity during the handover between the two distinct IFDS analyses is, however, non-trivial. Consider the example in Listing 29. In Line 7, the source value is obtained and assigned to a heap object. This triggers a backward alias analysis as shown by the dashed black line at the bottom in Figure 9. The taint is propagated back to Line 4 where the alias on `p2.f` is created. Since a new tainted access path has been discovered, the respective taint is handed back to the forward solver as shown by the upper dashed line in the figure. The taint is then propagated forward to Line 8 where the leak is correctly reported. However, the handover between backward and forward solver also leads to a false positive. Note that, to clarify the presentation of this false positive, we left out the forward edges from the handover onward for the true leak we just explained and do not show path edges either. For the false positive, it is important to understand that aliases are taints on their own. There is no distinction between an alias and a taint in the analysis. Therefore, after the handover, the access path `p2.f` is tainted from Line 4 onward, and, consequently, a false leak is reported in Line 5.

The problem occurs, because in reality, aliases are not standalone elements. Instead, all aliases of a given heap object (including what is represented by the original access paths) form an equivalence class that has a common state. Once one of the aliases is tainted, the heap object is tainted, and thus all of the aliases are tainted. In the analysis, we, however, have one state per alias, because each alias is represented by its own taint abstraction. In other words, when the analysis performs the backward propagation, it loses track of the information that the heap object is not yet tainted. The analysis thus needs a mechanism to only *activate* the taint abstractions that represent aliases when passing over the original line that tainted the heap object and for which the alias analysis was originally started. This would either require a different representation of aliases (one that is not equal to taint abstractions) or a linking between taint abstractions. Both are not trivial to integrate into the IFDS-based infrastructure and have drawbacks on their own. Firstly, having a different representation for aliases would prevent them from re-using the same IFDS problem for forward propagation after the handover has happened. Note that after a new alias taint has been discovered, it must be propagated forward to see whether additional aliases are derived from this newly discovered alias. As long as aliases are equal to taints, this can be done by the normal taint propagation problem. When using a different representation, this logic would have to be duplicated. For the Boomerang analysis, this is not an issue, because it is a standalone alias analysis and therefore cannot share abstractions or propagation logic with a client analysis such as FlowDroid in the first place. Secondly, linking the base taint abstraction with its aliases would violate the basic IFDS principle of local, point-wise propagation. In that case, when propagating over a statement, the flow function would not only have to be computed once for the incoming taint, but additionally once per associated taint, making data flow abstractions buckets rather than facts. It is unclear how this would work when aliases are injected into the forward solver at earlier statements at which the original taint was never propagated, such as in the example.

Instead of changing the representation of aliases or linking them with the taints from which they were originally derived, we therefore chose a simpler solution. Each alias taint can be associated with the *statement* at which the alias analysis was started. It becomes a taint on its own, but is inactive. Inactive taints that reach sinks are not counted as leaks, they are silently propagated onward. Only when the taint is propagated over the statement referenced in the abstraction (called the *activation statement*) it is activated can then again cause leaks. Technically, activating a taint abstraction simply means that the activation statement in the abstraction is reset to `null`. Afterward, the taint can cause leaks again as normal. In the example in Listing 29, this means that the taint passed to the backward solver in Line 7 (the one that goes across the lower dashed line in Figure 9) is inactive. It is associated with Line 7 as its activation statement. This information is kept as-is and propagated as part of the abstraction. When the taint is handed back to the forward solver (upper dashed line in the figure), the taint is still inactive and therefore, no spurious leak is reported in Line 5. Only when the taint is passed over Line 7, it becomes active again and can thus cause the correct leak in Line 8.

Each flow function in the forward solver checks whether it must activate the incoming taint at the current statement. For normal flow functions, this check is trivial, because it only needs to compare the current statement with the activation statement stored in the taint abstraction. For the call-to-return flow function, the situation is more complex, because it skips the callee and all of its transitive callees. It must therefore activate the taint, if the activation statement stored in the incoming taint abstraction appears in any of the methods of the whole skipped call tree. Technically, it must iterate over all callees of the current call site, and all of its transitive callees, and check whether the activation statement is reachable in any of those methods. Since this checks needs to be done whenever an inactive taint is propagated over a call-to-return edge, a naive implementation can cause a major negative impact on performance, especially for deep call trees. To mitigate this problem, FlowDroid maintains a cache with a mapping from activation statements to the set of call sites at which the respective taints must be activated when passing over. Note that this mapping is independent from any concrete taint and is only influenced

```
1  void onCreate() {
2    Data k = new Data();
3    Container c = new Container();
4    Container d = c;
5    foo(c, k);
6
7    k.f = source();
8    leak(d.data.f);
9  }
10
11 void foo(Container y, Data z) {
12   y.data = z;
13 }
```

Listing 30: Act. Stmts on Call-To-Return Functions

```
1  void onCreate() {
2    Data d = new Data();
3    d = id(d);
4    d.f = source();
5
6    Data e = new Data();
7    e = id(e);
8    e.f = source();
9  }
10
11 Object id(Object o) {
12   return o;
13 }
```

Listing 31: Activation Statements and IFDS Summaries

by the code structure. It is essentially an over-approximation of which statements are reachable from which call sites.

Computing this mapping for each statement that could potentially become an activation statement would, however, be costly. Therefore, FlowDroid exploits another observation. A alias taint is always propagated over the backward solver's IFDS return edge before it can be propagated over a forward call-to-return edge that can activate it. If the alias is created in a callee, then the data flow must first leave this callee in the alias analysis, before it has the possibility to arrive at the same call site again (e.g., due to a loop) and be propagated over it in the call-to-return function. This enables FlowDroid to populate the mapping of activation units to call sites on demand. Whenever the backward alias analysis leaves a method, the concrete activation unit in the current taint and the concrete call site at which the return happens are added to the cache. For a motivation as to why this approach is correct and does not miss entries in the mapping, consider the example in Listing 30. The tainted value is read from the source in Line 7 and written into a field `k.f`. The previously-called method `foo()` has already created an aliasing relationship between `k` and `c.data`. Since `c.data` is also aliased with `d.data`, a leak happens in Line 8. Note that the backward alias analysis must first finish the analysis of method `foo()`, and then return into the caller, before it can find the second alias on `d.data`. At this time, the alias is still inactive. Only when moving forward then and passing the call to `foo()` in Line 5 again, the taint is activated. The analysis can now exploit the cache entry that was created when leaving the `foo()` method earlier. In total, activation statements provide a simple and efficient solution that makes FlowDroid's context-sensitive alias strategy fully flow-sensitive as well.

On the downside, storing the activation unit inside the taint abstraction can also drag unwanted dependencies into the IFDS summaries. Consider the example in Listing 31. Taints are introduced in Lines 4 and 8. In both lines, the backward alias analysis is triggered, because the value obtained from the source is written into a heap object. In both cases, the statement right before the source call is a call to the same `id()` method. In both cases, the context is a taint on the first and only parameter of the method. Normally, one would expect the IFDS algorithm to analyze the `id()` method once, create a summary for it, and then apply this summary when processing the second call. With activation statements in the taint abstraction, this is, unfortunately, no longer the case. For the first call, the statement in Line 4 is the activation unit. Therefore, this abstraction is not equal to the abstraction at the second call that contains the statement in Line 8 as its activation unit. Technically, the two calls happen under different contexts and the summary generated for one context cannot be reused for the other one. Therefore, we make the use of activation statements optional and allow the context-sensitive aliasing strategy to be used without them if enabling them would negatively impact scalability for the current app at hand. Enabling or disabling activation statements is a trade-off between scalability (processing time and memory consumption) on one hand and precision on the other hand.

On Aliasing and Distributivity

Though the above IFDS-based alias analysis algorithm described above conveniently fits into FlowDroid's architecture and has proven to be precise in practice, aliasing is not a distributive problem and thus violates one of the preconditions of IFDS. By nevertheless formulating it as an IFDS problem, we conservatively over-approximate the

possible aliases of a given access path. One example in which this leads to a false positive is shown in Listing 32[20]. We left away the calls to the constructors of the local variables for the sake of brevity, these lines would not contribute any further insight into the problem at hand. In the example, a conditional decides between two possible pairs of aliasing relationships: Either (1) `x` and `a`, `y` and `b` are aliased, or (2) `x` and `p`, `y` and, `q` are aliased. A sensitive value is assigned to `q.b`. In the first case (the conditional evaluated to `true`), there is no alias to `q.b`. In the second case (the conditional evaluated to `false`), `y.b` must be tainted as well which then leads to `x.attr.b` being tainted in Line 13. Since we were in the second branch above (which is mutually exclusive with the first branch), there are no more aliases that must be tainted. More specifically, one cannot take the taint on `x.attr.b` and go back to the first branch of the conditional to derive a taint on `a.attr.b` in Line 6. There is no run of the program that goes through both branches of the conditional at the same time. A distributive framework such as IFDS, however, fails to keep the branches apart. It assumes that after the conditional, one is free to join the information obtained from each branch and continue with the union (or some other suitable merge result). When FlowDroid's context-sensitive alias analysis processes the assignment in Line 13, it will trigger a new backward propagation for `x.attr.b` to find potential aliases. At this point, there is no information that this taint is only valid under the assumption that the second branch was taken in the conditional, and so FlowDroid assumes that it is perfectly legal to process the first branch and join the results into whatever already exists in the taint set. Since this is a conceptual problem of applying IFDS to aliasing problem, it cannot easily be solved or circumvented. However, such cases are rare in practice and in our experiments we have not seen any major sources of false positives from this inaccuracy.

```
1  void onCreate() {
2    A b, q, y;
3    B a, p, x;
4
5    if (Math.random() < 0.5) {
6      x = a;
7      y = b;
8    }
9    else {
10     x = p;
11     y = q;
12   }
13   x.attr = y;
14   q.b = source();
15   leak(a.attr.b);
16 }
```

Listing 32: False Positive due to Non-Distributivity

4.8.5 Alias Analysis for Implicit Data Flow Analysis

When implicit flow tracking is enabled and a method is only called under a condition that depends on tainted data, then every assignment to a heap object inside that callee creates a new taint as explained in Section 4.6. This is because the state of the heap object then transitively depends on the tainted value. Recall that, for instance, writing a constant value into a field if and only if the user's password is *mySecretPwd* yields information about the password, namely whether it is equal to *mySecretPwd* or not. In other words, all fields that are written somewhere in a conditionally-called method are considered to be tainted upon return from that method. Non-heap objects need not be tainted, because their state is local to the callee and cannot be observed externally (aside from return values which are handled separately).
For the static data flow analysis, this gives a different view on the role of aliasing. In a traditional eager aliasing strategy, whenever a heap object is tainted, the alias analysis must find all potential aliases of this heap object and taint them as well. This leads to one alias query per statement that assigns a value to a heap object. Ideally, one wants an analysis that is context- and flow-sensitive. In the case of a conditionally-called method, this complexity is unnecessary. The task is no longer to find the aliases of some access path `a`, but to enumerate *all* aliases of all heap objects written in that method, because *all* of those heap objects will be tainted.

```
1  void onCreate() {
2    A a2 = a;
3    a.b.data = ``'';
4    if (source.equals(``mySecretPwd''))
5      foo(a);
6    sink(a2.b.data);
7  }
```

[20] The example was originally given by Uday Khedker from IIT Bombay in response to our presentation of FlowDroid's IFDS-based alias analysis and has been adapted for the purpose of this thesis.

```
 8
 9 void foo(A a1) {
10   A a2 = a1;
11   B b = a2.b;
12   b.data = ``Hello World'';
13 }
```

Listing 33: Aliasing for Conditionally-Called Methods

Consider the example in Listing 33. The analysis creates an implicit taint when processing the conditional in line 4. On the method call in line 5, an empty abstraction is passed into the callee `foo`. The assignments to local variables inside `foo` are skipped. Line 12 contains an assignment to a heap object, so the left side of the assignment gets tainted. Additionally, an alias query for `b.data` is started. Note that any of the alias analyses could now be used to resolve the query and find the taint on `a1.b.data`. With the knowledge that `foo` was called conditionally, the analysis can now, however, be faster. The key idea is that aliasing relationships in a method are static and independent of the concrete query or position in the code at which the query is performed. We can therefore improve performance by switching to a context- and flow-sensitive alias analysis.
Conceptually, one can therefore use SPARK for analyzing alias realtionships inside conditionally-called methods. SPARK has already been used to construct the callgraph, and consequently, SPARK's points-to analysis has also already been run, because it is integrated into SPARK's callgraph construction algorithm. On the downside, SPARK only provides an interface for binary alias queries and is not able to enumerate aliases as discussed earlier. Simulating this missing enumeration feature would lead to an algorithm similar to our PtS-based alias algorithm explained in Section 4.8.2. For technical reasons, we opted to not re-use the existing PtS-based algorithm, but create a very simplistic specialized implementation for conditionally-called methods that delivers improved performance. All the alias relationships are be pre-computed once per method and then cached inside a lookup table. Given such a lookup table, the alias analysis only needs to perform a fixed point iteration starting with a given initially tainted access path.

Access Path 1	Access Path 2
a2	a1
b	a2.b

Table 1: Aliasing Lookup Table for Method `foo`

Consider the lookup Table 1. If `b.data` is tainted and the lookup table states that `b` aliases with `a2.b`, then the taint `a2.b.data` gets added to the taint set. If `a2` aliases with `a1` according to the table, the taint `a1.b.data` gets added to the set. After that, no more new taints can be generated through table lookups and the alias analysis terminates. The taint on `a1.b.data` can then be passed back into the caller `onCreate` to continue with the normal taint propagation. In the caller, FlowDroid will automatically return to one of the other alias analyses to process further assignments to heap objects since it no longer analyzes a conditionally-called method. The lookup table-based analysis is never used outside of conditionally-called methods. Further, recall that in FlowDroid, the alias analysis is also invoked when returning from methods. In the example, the flow-sensitive alias analysis would, for instance, be used to then resolve that `a.b.data` in the caller aliases with `a2.b.data` which is finally leaked.
Note that this lookup table for aliases is inherently intra-procedural as well as flow- and context-insensitive. It is therefore possible to pre-compute the table once per method. If a method contains multiple assignments to heap objects, only the fixed point computation must be re-done per assignment, but not the lookup table construction. This is a key difference to, e.g., the flow-sensitive alias analysis presented in Section 4.8.4 which needs to perform a full propagation for every new alias query. Further note that there is no loss in precision or completeness when switching from the flow-sensitive alias analysis to the lookup table-based analysis in the case of conditionally-called methods.

4.8.6 Related Work on Alias Analysis

Several researchers have already looked into alias analysis, be it as a standalone problem, or specifically as a tool for enabling other analyses, including taint tracking. Many analyses have been proposed so far, such as the context- and flow-insensitive SPARK [84] analysis which is the default in Soot, or Paddle [83], its context-sensitive (but still flow-insensitive) counterpart. Doop [24] is a direct competitor of Paddle, designed to share exactly the same

logical points-to definition, and thus precision. Doop is based on Datalog combined with Binary Decision Diagrams (BDDs), an approach introduced to program analysis by Whaley et al. [143].
All these analyses, however, provide a very simple interface. They check whether two given variables or two given variable/field pairs alias. As explained above, such an interface is not a good conceptual match for a data flow tracker such as FlowDroid for two reasons. Firstly, it does not allow for querying all aliases of a given taint, but requires the data flow tracker to query all possible combinations. Secondly, it does not support the additional precision of having an access path with an arbitrary number of field dereferences instead of only a base object and a single field. De et al. [34] solve the second issue by providing a flow- and context-sensitive alias analysis for access paths. Notably, their analysis also supports strong updates. On the other hand, their work does not solve the first issue of enumerating aliases. The pointer analysis implemented in Andromeda by Tripp et al. [135] allows for such enumeration, but does not provide full flow-sensitivity, as it misses a concept akin to activation statements.
Our flow-sensitive alias analysis presented in Section 4.8.4 provides both compatibility with access paths and alias enumerations, and is context- and flow-sensitive. Furthermore, it supports strong updates at least on an intra-procedural level. Additionally, our analysis is, just like Andromeda, demand-driven. Aliases are only computed for those access paths that are actually tainted. The other approaches above use an ahead-of-time, whole-program approach. In data flow analysis, only a subset of all possible access paths is ever tainted. Consequently, an ahead-of-time approach is likely to compute many aliasing relationships that are correct, but irrelevant.
An earlier (though context- and flow-insensitive) approach to on-demand pointer analysis in the classical model (i.e., not based on access path and not enumerating) was presented by Heintze et al. [63]. Sridharan et al. [127] formulated the aliasing problem using a context-free language that can be solved on a graph representation. This approach is field-sensitive through labeling graph nodes, but flow-insensitive. DynSum by Shang et al. [121] uses summaries to improve the performance when answering multiple points-to queries within the same method. Our flow-sensitive alias analysis uses IFDS method summaries to store the effects of complete method execution on a given incoming context for later reference. The analysis by Yan et al. [148] is a pure alias analysis, as opposed to a full points-to analysis that can link objects to allocation nodes. They formulate the alias problem as a CFL-reachability problem use summaries for methods with a large number of callers. Boomerang by Späth et al. [126] was inspired by FlowDroid's context-sensitive alias analysis. It improves the analysis further and generalizes it. In contrast to the work presented here, their algorithm is no longer tightly integrated into the FlowDroid architecture using multiple solvers that need to inject context into each other. Instead, they provide a standalone pointer-analysis framework that, while it provides similar functionality, can be used with arbitrary clients. They even evaluated Boomerang by replacing FlowDroid's context-sensitive alias analysis with their new approach, showing that it reduces the number of alias queries the data flow analysis must issue by 29.4 times. Still, according to the measurements in the Boomerang paper, FlowDroid's original alias algorithm performs 1.6 times better than Boomerang with regard to total analysis time.

4.9 Library Call Handling

Programs usually rely on libraries to provide their functionality. The most widely used library is the Java Class Library (JCL) itself, but other third-party libraries exist as well. These libraries perform tasks such as cryptographic computations, graphics rendering, or database and file access. If a program creates an `ArrayList`, adds a element to it, and reads it back, it already exchanges (potentially sensitive) data with a library. A static data flow analysis on a Java program is thus incomplete if it does not take calls to library methods into account. In the example with the `ArrayList`, the analysis must know that adding a tainted element to the list and reading it back again yields tainted data. Not modeling the list would make the analysis lose the taint and potentially lead to a missed leak.
As an alternative, a static analysis can also merge the library code with the program under analysis and analyze both as a single entity. While this approach is conceptually simple, it can negatively affect performance. For many applications, the size of the library code (such as the complete JCL implementation) greatly exceeds the size of the application code. As a consequence, the analysis spends more time analyzing the library than the actual program of interest. Furthermore, this increase in the overall size of the code to be analyzed can also increase the time and memory consumption of the analysis up to the point of becoming infeasible on common hardware configurations. Additionally, merging library and application code requires all library code to be present on the analysis machine which is not necessarily the case if the application was programmed against a stub. The real library implementation might only be available on certain devices such as the target smartphones or servers. Lastly, one can observe that libraries are usually only changed infrequently, rending a complete re-analysis of the library code on every analysis run an unnecessary effort. Therefore, more lightweight models of the library code (in comparison to full implementations) are required.

In FlowDroid, such library models can be integrated through the `ITaintPropagationWrapper` interface. The basic idea behind a taint wrapper is to inject fake taints into the call-to-return flow function that jumps over the respective library call. Even if the library implementation is unavailable (i.e., there is no outgoing call edge from the respective call sites) or the library method is stubbed out (i.e., simply throws an exception instead of doing any real work as in the case of Android), the call-to-return-function is still invoked. It can therefore still query the taint wrapper for information on which outgoing taints shall be created given a certain incoming taint. Abstracting this handling through an interface allows us as well as other researchers to experiment with different library models. Usually, there is a tradeoff between the time required to compute the models and the time savings achieved by using the summaries instead of analyzing the full library implementation. The taint wrapper interface has the following functions:

- **getTaintsForMethod()** This method takes a call site and an incoming taint abstraction. It returns all taints that are valid after the control flow has returned from the library method. Note that this also models kill flow as it is upon the taint wrapper implementation to decide whether to pass the incoming taint on as part of its result taint set or not.
- **getAliasesForMethod()** This method queries the taint wrapper for may-alias relationships inside the library methods. It takes a call site and an incoming taint. From this information, it computes all access paths that may alias with the incoming one at the given call site. Note that this method is currently only used in combination with FlowDroid's flow-sensitive alias strategy (see Section 4.8.4).
- **isExclusive()** Some library models are guaranteed to have complete information about the data flows inside the respective callee. The wrapper is then called *exclusive* for these methods. FlowDroid will not conduct an own taint analysis inside an exclusive method even if its source code is available. Only the taints returned by the taint wrapper will be propagated onward over the call site. More technically, the IFDS call flow function gets deactivated for methods for which the taint wrapper is exclusive. Note that exclusivity also applies to the alias analysis, not only to the taint analysis.
- **supportsCallee()** This method takes a method as input and checks whether this taint wrapper has a model for that method. In that case, even if there is an implementation for the respective method available, no assumptions about it are made. This is important when applying code optimization techniques prior to the taint tracking. If the method, for instance, always returns a constant value, the code optimizer may propagate this value into the caller and remove the call. For a library method, such a code change can, however, break the semantics of the program. The analyzed method code may, for instance, only be a stub against which the program is compiled, and not a real implementation. In such a case, the call must be retained in any case such that the taint wrapper can then simulate the behavior of the real method implementation during the taint analysis.

Exclusivity is an important concept. When analyzing a Java program, one usually still needs to have the JCL (more precisely: the JRE's `rt.jar` file) on the Soot classpath. Without this file, the analysis would not be able to construct proper class hierarchies and would miss the declarations of the library methods. When including this file, the analysis must, however, be able to explicitly not enter those methods for which the taint wrapper can already provide a complete model. Otherwise, no time would be saved through the model. While Soot is already able to exclude certain classes, thereby banning the method bodies in that class from being loaded, this exclusion is too coarse-grained for library models. A taint wrapper must be able to exclude single methods from the analysis, potentially based on the incoming taint abstraction. This fine-grained control is provided through the exclusivity check.

4.9.1 Easy Taint Wrapper

FlowDroid provides a simple default taint wrapper called `EasyTaintWrapper`. This wrapper is based on a small number of rules that cover most cases, but are not necessarily precise. With these rules, the easy taint wrapper emulates the behavior of those static data flow analysis tools [67] that employ hard-coded rules of thumb. The easy taint wrapper is configured with a file called `EasyTaintWrapperSource.txt` that defines three different sets:

- **Included Prefixes** A prefix is usually a package name. The taint wrapper only applies its rules when processing a call to a method where the name of the callee's declaring class starts with a registered prefix.

- **Taint Generators** This set contains signatures of methods that can create taints. `ArrayList.add()` can, for instance, create a taint on the base object (the list) when called with a tainted parameter. Note that interface methods can also be declared as taint generators. In this case, all classes that implement the respective interface method are considered to match the taint generator. The same holds for abstract base classes.
- **Taint Killers** This set contains signatures of methods that kill taints. `ArrayList.clear()`, for instance, removes the taint from the base object. If there had been a tainted object in the list before, it is now guaranteed to be gone. Conceptually, a taint killer is a manually-defined unconditional sanitizer.
- **Exclusions** The method signatures in this list belong to methods to which the easy taint wrapper rules shall not be applied. This set allows to explicitly exclude methods that would otherwise be captured as their containing classes or packages are included.

Upon a call to `getTaintsForMethod()`, these lists are evaluated to compute the taints that are valid when the control flow returns from a call to a library method. The easy taint wrapper assumes that interactions between program code and library code is restricted to method calls. In other words, the program can only taint fields inside a library object by calling a method on that object. Afterwards, it can read that tainted data back by again calling a method on the object, or by directly reading the field. The latter is possible, because the easy taint wrapper taints the complete base object including all fields inside it (asterisk appended to access path). The following sequence of checks and actions is performed for each call to a library method:

1. **Include Check** If the declaring class of the callee method is not in the set of included prefixes, only the incoming taint is passed on, i.e., the taint state remains unchanged. The easy taint wrapper also supports an *aggressive mode* that disables this check.
2. **Kill Check** If the signature of the target method is in the kill list, an empty set of taints is returned. This explicitly refrains from passing on the incoming taint.
3. **Exclusion Check** If the target method's signature is in the set of exclusions, only the incoming taint is returned.
4. **Propagation Check** If the base object of a method call is tainted and the call site is a definition statement, the left side of the assignment gets tainted as well. Note that this rule applies to all methods as long as the include, exclude, and kill checks succeed. In other words, the easy taint wrapper approach does not specify all methods that exhibit this behavior, but, on the contrary, explicitly excludes those methods that act differently. The default assumption is that if an object contains tainted data, all method calls on that object can potentially return this data. Note that this rule does not distinguish between the base object being tainted and some access path rooted at base object being tainted. In either case, some tainted data is stored inside the object and assumed to be retrieved by any call on the object.
5. **Generation Check** The the signature of the target method is in the set of taint generators and one of the call arguments is tainted, a new taint is generated. If the target method is an instance method, the base object of the call gets tainted. If the call site is a definition statement, the left side of the assignment gets tainted as well.

The **propagation check** rule is very broad. It generates new taints on all calls that are not explicitly excluded. Therefore, this rule can quickly lead to over-approximation, negatively impacting precision and performance. Therefore, it is important to properly configure exclusions. Additionally, the easy taint wrapper contains explicit handling code for the two most common special cases. A call to `equals()` should never taint the base object, even if the parameter is tainted. If this method is implemented according to its specifications, it is free of any side effects. The easy taint wrapper exploits this fact to exclude all calls to `equals()` regardless of the type in which this method is implemented, i.e., also for classes that explicitly overwrite this method. Furthermore, we do not consider the return value of the `equals()` method as tainted unless implicit flow tracking is enabled. We also implement sepcial handling for the `hashCode()` method, because we assume that the sensitive information stored in an object usually cannot be reconstructed from its hash code. Therefore, even if the base object is tainted, the return value of this method remains untainted.
The easy taint wrapper also supports a *conservative* mode. This mode simplifies the generation check rule from the list above to taint the base object of the call regardless of which method is called. In other words, the every

call to a method with a tainted parameter taints the base object, not only for those methods in the set of taint generators, but for all methods. This mode allows for an implementation-agnostic conservative over-approximation of library behavior. Note that even this mode is incomplete when considering complex dependencies between multiple objects. Refer to Listing 45 in Section 6.1 for an example in which the easy taint wrapper still cannot find a leak, even in conservative mode. In short, calling a method on an object can only taint this object with the easy taint wrupper, but not another object to which the current one holds a reference. For such complex example, approaches such as StubDroid (see Section 6) are required.
For most programs and libraries, the easy taint wrapper is sufficient. By using interface methods as taint generators, a relatively small configuration file is able to cover most of the commonly used Java API methods. The default configuration file shipped with FlowDroid contains about 430 lines. On the other hand, this configuration file must be assembled manually, making completeness guarantees hard to achieve. A fully automated and, additionally, more precise library model that can be plugged into FlowDroid's taint wrapper architecture is provided through StubDroid. We will present StubDroid in Section 6.

4.9.2 Library Detection

Modeling libraries only works under the assumption that a clear distinction between program code and library code can be efficiently made. For those libraries that are pre-installed on the target device, the distinction is easy. The respective package and class names are known in advance, and are unlikely to ever change as backward-compatibility is usually a key requirement for the vendors. Furthermore, the analysis does not find the respective code inside the program, but on external locations somewhere on the classpath. Some libraries are, however, not pre-installed on the device, but rather shipped with every program that uses the particular library. Android's support library, for instance, emulates some features of newer Android operating systems on devices running older Android versions to ensure backward-compatbility for programs. Depending on the features an app uses, the corresponding version of the support library is packaged into the app's APK file during compilation. Due to this re-packaging, an analysis cannot distinguish between app code and library code based upon the location from which the class or method in question was loaded. Libraries can be loaded from additional JAR files on the Soot classpath as well as from the app package (APK file) itself. Still, in the simple re-packaging case, identifying libraries based on package and class names is sufficient.
If the app developer, however, runs an obfuscation or optimization tool on his app after compilation, the package and class names are changed to shorter versions. These do not bear any further semantics such as `a.b.abc` instead of `android.app.LibraryClass`. A library model that expects the original class name will thus fail to recognize the renamed version of the re-packaged library. Though the library code remains the same, the analysis will act as if no library model were available for it. Recall that this is only an issue if the library code is compiled into the app. Therefore, the analysis will not miss any leaks despite the renaming, because it can always analyze the full library code together with the app. Still, this can greatly increase the size of the code to be analyzed and lead to exactly those performance issues that the external library models were originally designed to solve. Identifying obfuscated libraries is an open challenge and orthorgonal to the work presented in this thesis.
Other approaches such as AdSplit [122] propose changes to the Android framework to separate apps from their libraries and avoid such code mangling. In AdSplit, advertisement libraries (which are common cases of library use on Android) are run as separate apps inside separate processes. While such changes are hard to make to an existing framework such as Android, it could also allow for separate sets of permissions for the app and its libraries which would enhance security. For FlowDroid, this would allow the app and its libraries to be analyzed as totally separate entities. This would also help capture the semantics of a library: No matter where a library X is used, it always exhibits certain behavior on its inputs, such as leaking incoming data to the Internet. StubDroid (see Section 6) uses a similar reasoning on the level of methods: A library method behaves identically in terms of data flow regardless of the program in which it is used[21].

4.9.3 Library Stub Generation

FlowDroid is used as a component in many other research projects. In some of these projects, developers rely on the callgraph alone. They choose FlowDroid to take advantage of its Entry Point Creators (see Section 4.15) so that they do not have to model, e.g., the Android lifecycle on their own. This callgraph is, however, partially incomplete for library callbacks. Recall that taint wrappers define how data flows are propagated through library

[21] See Section 6 for additonal considerations on callbacks, etc.

methods for which no implementation is available. A taint wrapper, however, does not extend the callgraph. If there is no implementation for the library method, i.e., no outgoing call edge on a call site that invokes a library method, the taint wrapper will not add the missing callgraph edge. For the taint analysis, this makes no difference, as the respective taints are still injected in the call-to-return function due to the external model. For the callgraph-only external user, it can lead to false negatives. If the developer takes the callgraph created by FlowDroid and enumerates the reachable statements, he will, for instance, not find all statements inside custom thread implementations. The `java.lang.Thread` class is part of a library (the JCL for Java programs and the Android framework for Android apps) and usually excluded from the analysis for performance reasons. If the code under analysis calls `Thread.start()`, the callgraph generator fails to create outgoing edges for this call, let alone map this call to the `run()` method of the custom thread implementation.
As a solution for this challenge, FlowDroid provides an additional concept besides taint wrappers. Similar to the mocks proposed in DroidSafe [60], FlowDroid patches some library classes with mock implementations. Instead of simply replacing the library with a custom implementation altogether, it dynamically creates the respective code. That has the advantage that the user is still free to choose whether to run FlowDroid against a full or partial implementation of the library or none at all. The tool detects the missing parts and adds mock implementations for just these parts, essentially leading to a merged library. Obviously, this approach shares the very same issues of the DroidSafe mocks: Developing the mocks is a major undertaking. Therefore, FlowDroid only generates mocks for those cases in which problems with the callgraph have been reported. As of now, these are a handful of methods from `java.lang.Thread` and Android's `android.os.Handler` class.

4.10 Native Call Handling

Java programs and Android apps can both interact with native, platform-specific code written in unmanaged languages such as C or C++. The Java VM and Android's Dalvik VM both support JNI, the Java Native Interface. The JNI allows a program running inside a Java (or Dalvik) VM to load a native library and call methods inside this library. The library can also use JNI to access classes, methods, and fields inside the Java or Dalvik code. This allows the programmer to write arbitrary parts of his program in native code with practically no limitations on the interactions between both worlds. Programmers mainly use this feature to re-use existing native libraries through encapsulation. The native crypto libraries present on many operating systems are a good example for such practice. The Soot compiler framework, however, is not capable of processing native code, neither as source code nor as binaries. Additionally, it is unclear whether native costructs such as pointer arithmetics can be expressed in the Jimple intermediate representation without major enhancements to the language itself. Therefore, a tool such as FlowDroid which is based on Soot is unable to conduct a data flow analysis on native code.
In general, the same considerations that apply to library methods also apply to native methods, with the only difference that there is no choice as to whether to analyze them or not. Instead, FlowDroid must rely on external data flow models for these methods. According to our prior research [111], only 14.5% of all Android apps use native code. The number is in line with the findings of other researchers (15% of all apps with less than 50,000 downloads use native code according to Viennot et al. [140]). Especially on Android, the use of native code is usually centered around computation-intensive operations such as graphics or video rendering. Therefore, native code is also more prevalent in the big, popular apps that have been heavily engineered and optimized. According to Viennot et al., 70% of all apps with more than 50 million download make use of at least one native library.
Luckily, these performance-critical operations commonly performed in native code do not typically process sensitive data. Therefore, excluding (i.e., under-approximating) custom native methods does not affect the results of a data flow analysis for the average Android app. Still, this leaves an attacker with the possibility to deliberately hide data flows by transforming data in native code, or simply by calling a native method that contains the full data leak. As explained above, a native method has full access to the Dalvik part of the app: It can, for instance, call the source and directly leak the data, which keeps the data flow completely outside the Java/Dalvik part of the code. In this case, not even consernatively over-approximating the behavior of custom native method would detect the leak. For the purely Java- or Dalvik-based static data flow tracker, no tainted data is ever passed into the native code and no data is ever returned from the native code to the Java or Dalvik part. Therefore, combining FlowDroid with external approaches for analyzing native code is an interesting area of future work. The models created by the external tool would then be parsed and applied by FlowDroid's native call handler. The ROSE [108] compiler framework, for instance, contains a static data flow tracker for native code, though this approach cannot work on binaries. It requires the source code of the native library. Still, even such a white-box approach is helpful when dealing with common open-source libraries embedded into Android apps. Alternatively, techniques for conducting a dynamic data flow analysis on JNI code have been proposed in the literature [107].

```
1  void test() {
2    String s = source();
3    Object o1 = s;
4    Object o2 = new A();
5    leak(convert(o1));
6    convert(o2);
7  }
8
9  String convert(Object o) {
10   return o.toString();
11 }
```

Listing 34: Context-Sensitivity Example

```
1  void test() {
2    A a = source();
3    Object o = a;
4    if (random() < 0.5) {
5      A a2 = (A) o;
6      leak(a2.data);
7    }
8    else {
9      B b2 = (B) o;
10     leak(b2.data);
11   }
12 }
```

Listing 35: Type Propagation Example

Even without considering custom native code, some native methods are directly contained in the Java / Dalvik runtime. The most commonly used example is `System.arraycopy` which copies a range of elements from one array to another. Without modeling this native method correctly, data leaks can easily be missed even if the developer does not deliberately hide malicious behavior in native code. As a manual investigation of popular Android apps and Java programs shows, these system-defined native methods are, however, few. In FlowDroid's default implementation of the native call handler, we therefore opted to include hand-written models for the behavior of these methods.

4.11 The Effect of Data Type Propagation

FlowDroid invokes its entry point creator (see Section 4.15) to create a dummy main method which serves as the entry point for callgraph construction. The callgraph construction in itself is done through one of Soot's built-in callgraph algorithms, depending on the FlowDroid configuration. These algorithms include CHA, VTA, RTA, and SPARK with full precision. All of these callgraph algorithms are context-insensitive. In a number of cases very common to Java programs and Android apps, this can lead to imprecision as shown in Listing 34. The original data was of type `java.lang.String`, but at the call site in Line 10, the variable is only declared as `java.lang.Object`. A context-insensitive analysis such as SPARK can only propagate all possible types to this position and merge them into a single list. Therefore, the call to `toString()` is assumed to go to at least `String.toString()` and `A.toString()`, because both `String` and `A`-typed objects arrive at that call site. If the method `A.toString()` leaks its parameter value, this would lead to false positive. The method `A.toString()` is never called with any tainted data, but due to the context-insensitive callgraph algorithm, the analysis cannot distinguish that the sensitive data only goes to `String.toString()`, and the non-sensitive data only goes to `A.toString()`. A context-insensitive analysis cannot distinguish that there are different callee methods at line 10 depending on the calling context and instead merges all of these contexts, leading to the described false positive.
On possible solution would be to switch to a context-sensitive callgraph algorithm as discussed in Section 4.2.6. In this section, we discuss an alternative approach that also aims at pruning impossible edges from the callgraph, but without the high construction-time overhead required for context-sensitive callgraphs. Concretely, FlowDroid propagates the types of tainted variables as part of the taint abstraction, more precisely: as part of the access path. In other words, an access path not only consists of a base object (that is potentially `null` in the case of static fields) and an array of fields, but also of a base object type and an array of field types. Note that this information is not redundant, because the additional type information can be more precise than the statically-declared types of the base object or fields. In the example in Listing 34, the statically-declared type of variable `o2` is `java.lang.Object`. The propagated type information, however, stays `java.lang.String` which is the type of the data returned from the source, even if there are implicit downcasts in the source code as in Line 3. The propagated type information is only adapted if there is an explicit cast to more precise type.
This type information can then be used for pruning infeasible callees at call sites. In Line 10, the base object `o` of the method call is tainted. The access path of the taint has `java.lang.String` as its propagated base type. The callgraph still contains outgoing edges to `A.toString()` and `String.toString()` and the IFDS call flow function is called for both possible callees. FlowDroid, however, detects that `A` is not cast-compatible to the propagated type `java.lang.String` and therefore kills all taints on the call flow function for callee `A.toString()`. In other words, the flow functions prune the over-approximated context-insensitive callgraph according to propagated types. Note

that the implementation for this pruning is done in the *Typing Propagation Rule* which is active by default, but can also be disabled on user request.
Precise type information on access paths further allows FlowDroid to prune taints on incompatible operations. Take the example in Listing 35. The call to the source method in Line 2 returns an object of type `A`. The source is configured such that `a.*` is tainted, i.e., not only the object itself, but also all of its fields. The object is then cast down to `java.lang.Object`. In Lines 5, and 9, the object is cast back to type `A` and `B` respectively. The cast to `B` will always fail, because class `A` is not cast-compatible to class `B`. Therefore, the sink call in Line 10 can never be reached. All control flow into this branch of the condition will fail with a `ClassCastException`. Only the call to the sink in Line 6 leads to a leak. With the type information in the access path, the *Typing Propagation Rule* in FlowDroid can check every typecast. If the target type of the cast is not cast-compatible with the propagated type in the access path, the taint along the current control flow path is killed. This avoids a false positive when analyzing Listing 35. Note that code similar to the one in Listing 35 is common if code deals, e.g., with serialized data. After deserialization, the declared type is always `java.lang.Object` and the code must use `instanceof` checks and typecasts to retrieve the different data objects.
One limitation to this approach is that precise type information is only available for tainted objects. If the tainted data is passed through a parameter and the base object itself is not tainted, no propagated type information for that base object is available. In such a case, FlowDroid cannot filter the callgraph edges and must assume that all call targets are possible. Therefore, the type propagation does not increase precision as much as a full context-sensitive callgraph algorithm would. On the other hand, the type propagation does not induce any overhead into the callgraph generation. The overhead for checking for spurious callgraph edges or impossible typecasts is negligible. Therefore, we propose type propagation as an alternative to the classical approach of scaling callgraph precision vs. callgraph generation performance. We report on the impact that this feature has on the performance of the analysis and the number of leaks that are detected in Section 8.6.
Recall that IFDS uses method-level summaries to improve the performance as explained in Section 3.4. When augmenting access paths to contain type information, this, however, decreases the reusability of the IFDS method summaries. Consider again the example in Listing 34. If the access path does not contain any type information beyond the declared types, the IFDS solver will create a single summary for method `convert()`. This summary maps the data flow fact on the first and only parameter to the data flow fact on the return value. The summary can then be applied to all call sites of method `convert()` that pass in a tainted object as the first and only argument. With types in the access path, a single summary is no longer sufficient. If the method was originally called with a string argument, the resulting summary is only applicable to other call sites that also pass in a string. If a call site passes in some other object, the incoming taint abstraction is not equal to the abstraction for string and the summary is thus not applicable, requiring the IFDS solver to re-analyze the method. In cases in which the behavior of the method is independent of the incoming data types, splitting the summaries according to types induces extra computational cost. This leads to a tradeoff between the savings of not having to analyze spurious callees as explained above, and the extra cost of not being able to apply certain summaries. Note that simply ignoring type information when applying summaries is unsound. As explained above, when analyzing a method body, the type information is used to kill taints at impossible type conversions. In the code in Listing 36 this leads to two semantically different summaries based on the input type. Only if the argument (which is declared as class `Foo`) is of type `Bar`, the result is tainted as well. If the argument is of a type that is not compatible with `Bar`, the return value is not tainted. A single, type-agnostic summary could not capture this behavior. Therefore, there is no simple solution to the tradeoff between more summaries and increased call target precision presented above. This issue is related to storing the activation statement in the taint abstraction to ensure that FlowDroid's context-sensitive aliasing algorithm is flow-sensitive as well as described in Section 4.8.4.

```
1 String typeDependent(Foo o) {
2   if (o instanceof Bar)
3     return o.toString();
4   else
5     return null;
6 }
```

Listing 36: Type-Dependent Method Summaries

4.12 The FastSolver: An Optimized IFDS Solver

Since FlowDroid propagates millions of taint abstractions for average-sized Java programs or Android apps, the performance of the core IFDS solver is very important. This solver must only require a minimal amount of time and memory per propagated flow fact. Originally, FlowDroid was built on the generic Heros [23] IFDS/IDE solver. For large problems, this solver, however, proved to be inefficient. Additionally, by relying on Heros, the possibilities for problem-specific enhancements to the solver were limited as Heros is a general-purpose solver library. Therefore, we opted to derive our own solver, called *FastSolver*. While FastSolver is a drop-in replacement for Heros to allow for A/B-testing our new solver against Heros, it is limited to analysis problems that share some characteristics with FlowDroid. The design of FastSolver is based on the following observations:

1. FlowDroid is only based on IFDS, not IDE.
2. FlowDroid never needs access to the graph of IFDS abstractions after they have been propagated. There is never a query such as "give me the computed abstractions at statement n for context d_1." Leaks are directly detected and recorded while evaluating the flow functions.
3. Flow functions can be complex. Their outputs are semantically correct, but not always minimal or optimally efficient.

The remainder of this section explains optimizations we derived from these observations that reduce the memory consumption and improve the performance of the FastSolver in comparison to Heros.

4.12.1 IFDS vs. IDE Solving

Internally, Heros is not a native IFDS solver, but a solver for the more general IDE [120] problem. It provides IFDS support by encapsulating IFDS problems as a special case of IDE. While this is conceptually valid and produces sound results, it makes use of an engine that is not tailored to the specific problem at hand. The IDE engine must keep state and perform computations to provide for features that are never used if only IFDS problems are considered. We therefore designed FastSolver as a native (i.e., non IDE-based) IFDS solver.
IFDS is a fixed point algorithm on the exploded supergraph. Whenever a new taint abstraction is generated, the solver needs to check whether this abstraction has already been seen for this context at this statement. If not, the taint abstraction is new and needs to be propagated to all of its successors in the interprocedural control flow graph. On the other hand, if it has already been seen, the taint propagation has reached a local fixed point and this taint abstraction does not need to be propagated onwards. Fomally, this means that for every pair of context and statement $\langle d_1, n\rangle$, a set of data flow facts $d_2 \subset D$ needs to be stored. Technically, this can be represented using a set of fact tuples $F_{ifds} \in \mathbb{P}(\langle d_1, n, d_2\rangle)$. There is only one performance-critical operation that this set F_{ifds} needs to support which is containment check and union for detecting the fixed point (see observation 2). In the implementation, this corresponds to the *add* operation of a hashset. This operation is executed once for every newly created taint abstraction.
For IDE, the data storage is more complex. One additionally needs to store a set of edge functions for each context, statement, and data flow fact. Therefore, one needs to store a set of function tuples $F_{ide} \in \mathbb{P}(\langle d_1, n, d_2, g\rangle)$. When emulating IFDS using an IDE solver, the function g is always the identity function. Therefore, this additional element in the tuple is wasted. Given that it needs to be stored for every combination of context, statement, and data flow fact, this is a considerable memory waste. Additionally, at the scale of millions of abstractions (and thus tuples F_{ifds} or F_{ide}), the performance penalty for maintaining the data structures that hold the tuples (e.g., growing a hash set) is notable. Therefore, storing as few elements as possible is vital.
Additionally, IDE contains a second phase which evaluates these functions g on the graph obtained from the initial propagation of data flow facts. The edges of this graph would be, in the notation from above, $e \in \mathbb{P}(\langle d_1, d_2\rangle)$ for every statement n. To be able to evaluate these functions, one needs to traverse the taint propagation graph. With a plain set representation F_{ide}, this is not efficiently possible. Therefore, additional lookup maps are required. In the Heros implementation, F_{ide} is actually represented by three different maps which, in turn, again contain maps to provide the three levels of indirection that arise when allowing queries on F_{ide}. For data flow tracking, this phase is not necessary. Recall that the functions g are always the identity function. By tailoring the solver at IFDS and dropping the IDE support, these lookup maps are no longer required, and can be replaced by one flat set. This cuts the overhead memory usage (maps, map entries, etc.) spent on storing the taint abstractions by about one third. As explained above, it also greatly reduces the time spent on managing data structures.

Additionally, the FastSolver supports a *memory manager*. The memory manager is an interface that allows clients of the solver such as FlowDroid to specify additional operations that are applied to new data flow facts before they are propagated onwards. In other words, the abstractions returned by the flow functions are passed to a central component that can modify or drop them before they are actually propagated by the IFDS engine. The memory manager feature is a consequence of observation 3 from above. It enables flow functions to focus on semantics, rather than performance details. In the following, we describe some of the concrete actions the FlowDroid memory manager takes.

Access Path Pooling

The FlowDroid memory manager caches access paths. Recall that FlowDroid is a flow-sensitive data flow analysis tool, i.e., must store the taint state per statement. If a certain access path is tainted at one statement, it is, however, likely that the same access path is also tainted at other statements in the same method. The access path cache allows to re-use the same object in this case, reducing the memory footprint of the analysis. Note that we only cache access paths, not full taint abstractions. When combining access paths with the additional information introduced by e.g., the alias analysis (activation statement) or the path reconstructor (links to predecessors and neighbors, see Section 4.13), the performance improvements gained through the cache are smaller than the additional overhead introduced through the cache management.

Abstraction Stripping

The flow functions always record the current statement when creating a new taint abstraction. The allows to later reconstruct the taint propagation path. If path reconstruction is not enabled, the memory manager removes this information before propagating the abstraction onward. This simplifies the abstractions without adding further complexity to the flow functions.

Abstraction Path Compression

As explained in Section 4.14, the taint abstractions do not directly reference the source from which they were originally derived. Instead, they point to their respective predecessor to construct a taint propagation graph. To keep the flow functions simple, every change to a taint abstraction is modeled as constructing a new taint abstraction with the changed data item. This new abstraction then points to the original one as its predecessor. Keeping taint abstraction objects immutable and allowing only to derive new ones from them makes sure that we can store the abstractions in sets without changing the hash code of an object after it has been put into the set. On the downside, this means that every single change to a taint abstraction creates a new object and increases the length of the current path through the taint graph. To resolve this issue, the memory manager compresses predecessors: If a taint abstraction `a1` was the input to a flow function and abstraction `a2` was its output, the predecessor of `a2` is forced to be `a`, skipping over all the intermediaries. Since these intermediaries are no longer referenced anywhere in the taint graph and have never been propagated on their own (recall that we dealing with the output of a single flow function application), they will be garbage collected by the JVM. This results in a reduced memory consumption as well as smaller taint graphs.

When the user is not interested in the full propagation path, but only wants to retrieve the source-to-sink connections, the taint graph can be compressed even more. In this case, the abstractions inside callees are not interesting, only the outcome of the method call is of interest to continue with the taint propagation. Consequently, when returning from a method, the memory manager sets the predecessor of the abstraction at the return site to be the incoming taint at the call site. This simulates that the callee is an atomic operation on the taint. When the taint finally reaches a sink, the backwards search then has to process fewer nodes before it finds the source, speeding up the process.

The memory manager also checks whether the new predecessor is the same object as the current abstraction. This can happen if a flow function just passes on the current abstraction without changing anything. Especially in the case of method invocations, the reduction can also yield such a case: The intermediaries might have been different, but in total, the method might be an identity function. The memory manager then passes on the original object, leaving not even a single distinct abstraction for the method call.

Semantic Validation

The memory manager is also a good central place for performing basic sanity checks on the generated abstractions. While walking up the predecessor chain of an abstraction, one should, for instance, never run into a loop. If a

loop occurs, this means that a taint is transitively dependent on itself which does not add any useful information to the overall taint analysis problem. In FlowDroid's memory manager, we detect such cases and report them to the log file. In most cases, loops are caused by bugs in the implementation of the flow functions or a custom implementation of one of the many interfaces (taint wrapper, native call handler, etc.)

4.12.3 Memory Thresholding

Since the IFDS solver performs a fixed point iteration on the exploded supergraph, the set of function tuples $F_{ifds} \in \mathbb{P}(\langle d_1, n, d_2 \rangle)$ is constantly growing. The solver must keep references to all these combinations of context, statement, and data flow fact, to properly detect when a fixed point has been reached. Not storing one entry could mean that, if there is a loop on the respective statement, the same fact is computed over and over again and the analysis does not terminate, because it simply cannot detect that the outcome of the respective flow function is not new, but has already been processed. On the other hand, keeping references to these tuples means that the respective taint abstractions d_1 and d_2 can never be garbage-collected by the Java Virtual Machine (JVM) at any time during the analysis. This in turn steadily increases the memory consumption of the analysis over time.
As long as enough free heap space is available, this is not an issue. Once the JVM starts to run out of heap space, it will perform more aggressive garbage collection and try to find and clean up objects that are no longer referenced. Note that there are, as in any other Java program, many intermediate objects that do not have "eternal" references to them. In some situations, the analysis can, however, not realistically find enough memory to store all the different taint abstractions. Assume that the program under analysis is very large and complex (i.e., passes tainted data through many different objects), and that a long maximum access path length has been configured. In this case, there are more different tainted access paths than what can fit into memory on most machines. Consequently, the analysis will at some point exhaust its available heap space. With the default configuration, the JVM will, however, not directly abort the process. Instead, for every newly allocated object, it will try to free up just enough space to fit this new object. Therefore, every new allocation suffers from a garbage collection overhead that greatly exceeds the time required for an allocation under normal (i.e., non-memory-pressured) circumstances. This factor is called the garbage collector's overhead, or *GC overhead* for short. The JVM will only abort when the GC overhead has reached a limit that makes it unlikely to be able to allocate even smaller objects anymore. In total, a FlowDroid analysis will appear "stuck" to the user, while in fact, it is mainly spending most of its time on garbage collection instead of on actually processing the data flow analysis.
From a user's point of view, this behavior is undesirable. One solution would be to decrease the GC overhead limit using the respective JVM parameter. This will, however, make the JVM terminate the analysis process and no results will be available to the user. If such a termination happens, a Java program has no possibility of saving its results or state (let alone that it would not have any free memory left to conduct any such operations). Therefore, we have extended FlowDroid with a *Memory Thresholding* mechanism. This mechanism uses the memory management beans available in the JVM to be notified when the percentage of free heap space left in the JVM drops below a certain threshold. If this happens, the IFDS solver is instantly aborted by converting all flow functions into *kill* functions. Afterwards, the global set of function tuples is set to `null`, allowing it to be garbage-collected together with all taint abstractions that are not referenced by any other object. This frees enough memory to allow for building taint propagation paths on the already-discovered results as explained in Section 4.13. Note that all abstractions that reach a sink are stored in a different set, so they and their respective predecessor chains are still available, even when the global set of function tuples is garbage-collected. This mechanism allows FlowDroid to supply the user at least with a subset of the results, even though the complete analysis could not be finished.
From the technical side, it is important to note that this approach, while effective in practice, is a race condition by design. When the threshold warning is triggered, the solver's executor threads, which compute the flow functions and propagate the data flow abstractions onward, are still active and execute concurrently. In other words, while the memory warning is processed and the flags to stop the IFDS solvers are being set, more memory is being allocated. In the worst case, if processing the warning takes longer than using up the remaining heap space, an `OutOfMemoryException` may be thrown before the corrective measures are applied[22]. To counter this problem, the threshold must be set low enough so that enough memory is still available for the intermediate processing. Setting the threshold too low, on the other hand, leads to the warning being triggered too early, while enough memory to continue and maybe even finish the analysis is still available. In practice, setting the threshold to 90% of the total JVM heap size seems to be a reasonable tradeoff.

[22] In practice, the risk is more that the analysis gets "stuck" in excessive GC cycles, but the effect of rendering the thresholding ineffective is the same.

4.12.4 Flow-Insensitive Solver Variant

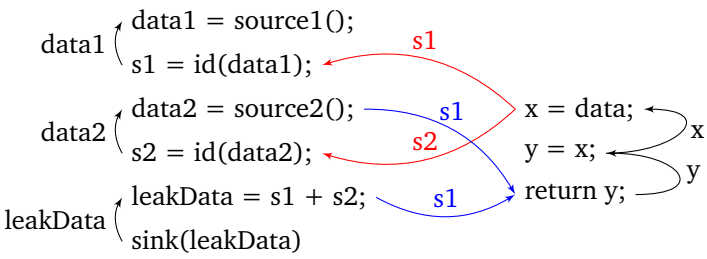
When conducting a flow-sensitive IFDS analysis, the solver needs to store each data flow abstraction at each statement at which it holds. On this mapping, the solver can check whether it has reached a fixed point. If a flow function computes a fact at a statement for which a fact equal to the new one has already been registered, the new fact is discarded. At some point, no previously-unseen facts are generated anymore and the solver terminates. Maintaining this list of tuples $F_{ifds} \in \mathbb{P}(\langle d_1, n, d_2 \rangle)$ (recall that the context d_1 under which the data flow fact holds must also be stored) can lead to memory exhaustion, becuse it requires many objects to be kept in memory for the full runtime of the solver. One possibility to simplify (and thus reduce the size of) this set of tuples is to eliminate elements. FlowDroid therefore supports a flow-insensitive, but context-sensitive variant of the solver. In this implementation, the tuple stores the method m containing the statement n, but not the statement n itself. Semantically, data flow facts hold for complete methods in this solver variant instead of for individual statements inside the method.
This also changes the semantics of taint propagation. Normally, when a new flow fact is generated in IFDS, this flow fact is then used as the input of the flow functions associated with all successors of the current statement. In other words, taints are propagated over the edges of the control-flow graph. In the flow-insensitive variant, a flow fact always holds for the entire method. Therefore, the flow functions associated with all statements in the current method must be evaluated on the new flow fact. Each fact generated for any of the statements in the method then again holds for the whole method and is again used as an input to all flow functions for all statements until a fixed point has been reached. This technique is obviously flow-insensitive. It reduces the size of each tuple F_{ifds} by one third. On the other hand, it can never kill taints inside a method, because there is no semantic for defining a state before and after the kill. This may lead to additional, unnecessary taints to be propagated, which can decrease performance. Secondly, there is no notion of reachability inside a method. Normally, flow functions only need to be evaluated for a certain flow fact if the statement with which the flow-function is associated is reachable from the statement that originally introduced the data flow fact. This constraint is no longer valid when applying flow facts to whole methods, which can, again, decrease performance. In summary, the performance tradeoff is only positive if evaluating a flow function is cheap, but storing copies of data flow facts is not. We evaluate the impact of using a flow-insensitive data flow solver on performance and memory consumption in Section 8.8.

4.13 Building Taint Propagation Paths

The description of the FlowDroid static data flow tracker has so far focused on propagating taint abstractions through the target program. Once a taint has reached a sink, a leak must be reported, including the original source of the tainted data and the sink where the leak was detected. Optionally, users must be able to also retrieve the taint propagation path, i.e., the sequence of statements over which the taint was propagated between source and sink. A trivial approach would propagate the source information as part of the taint abstraction. This allows the analysis to trivially extract the source at any time during the taint propagation, not only at the sink.
The drawback of this simplistic approach is, however, that taints from two different sources can never be merged during the taint propagation. Consider the example in Listing 10(a). The data stored in the variables `data1` and `data2` is obtained from two different sources and then flows into the same method `id()` which is essentially an identity function. Then, the two strings are concatenated and leaked. If the source statement is part of the taint abstraction, the `id()` method is called in two different contexts. Therefore, it must be analyzed twice, though the effect of the `id()` method on the tainted data passed in as a parameter does not depend on the source. Conceptually, this context splitting based on the source is unnecessary. Note that this additional analysis can severely negatively affect the performance of the overall taint analysis. Not only the callee, but the whole call tree rooted at it must be analyzed multiple times. Without the artificial splitting, this whole tree would only be analyzed once. Afterwards, an IFDS method summary for the callee would be available which can then be applied when processing the next call to that callee.
To solve this issue, we propose not storing the source information in the taint abstraction. The core idea is to make taint abstractions (and thus method call contexts which are modeled through incoming taint abstractions) indepedent of all information that is not required for the actual taint propagation. This maximizes the re-usability of the IFDS method summaries. Still, when a taint abstraction reaches a sink statement, the analysis must be able to recover the original source information. One approach that achieves both goals would be to switch from an IFDS analysis to one formulated in the more general and more expressive IDE framework. In the first IDE phase, the analysis could then leave the taint source out of the individual taint abstraction and generate a source-agnostic taint propagation graph. Over this graph, it could afterwards then propagate the concrete source information during the

```
1  String id(String data) {
2    String x = data;
3    String y = x;
4    return y;
5  }
6
7  void onCreate() {
8    String data1 = source1();
9    String s1 = id(data1);
10
11   String data2 = source2();
12   String s2 = id(data2);
13
14   String leakData = s1 + s2;
15   sink(leakData);
16 }
```

(a) Path Reconstruction Example



(b) Taint Graph for Example

Figure 10: Path Reconstruction - Example as Code and Taint Graph

second phase of IDE. The main drawback of this approach is that it is incompatible with the optimization discussed in Sectionřefsec:FlowDroid:fastSolver. These optimization are are mainly based on leaving away the additional lookup tables for jump functions that one has to maintain when running a full IDE solver. Additionally, building FlowDroid on an IDE solver would always lead to this overhead and would not leave room for configurable tradeoffs between performance and precision such as the ones we present later in this section using our different path builders.

Instead, we propose to keep IFDS as the underlying framework, and chain each taint abstraction to its predecessor. This technique implicitly builds a graph from the current abstraction upward to the source, an approach inspired by the work of Lerch et al. on FlowTwist [82]. Let us for now assume that the control flow is strictly linear without any join points. Only the taint abstraction at the source statement contains information about the source. All abstractions derived from it only point to their respective predecessor. Two abstractions `a1` and `a2` are equal if they model a taint on the same value, regardless of their predecessor. In the example in Listing 10(a), this means that incoming abstraction for the call at line 9 (i.e., the calling context) is equal to the incoming abstraction for the second call to the `id()` method at line 12. Therefore, the IFDS summary generated on the first call can be re-used on the second call.

For handling join points, having one single predecessor for each taint abstraction is not sufficient. In line 14 in the example, two tainted values are concatenated. The result is a combined taint on variable `leakData`. This new taint would normally have two predecessors. Recall that taint propagation in IFDS is point-wise. When the taint on `s1` arrives at line 14, the taint on `leakData` is first created. When the taint on `s2` arrives, the flow function again creates a taint on `leakData`. When attempting to propagate this second taint onwards, the IFDS solver, however, detects that an equal taint on `leakData` already exists. Therefore, the second taint on `leakData` is not propagated onwards which is again a saving in comparison to propagating independent taints for every source. Instead, the second instance of the taint on `leakData` is registered as a *neighbor* of the first taint on that variable. More generally, each taint has a link to its predecessor plus a (possibly empty) set of neighbors. A neighbor is an equal node (i.e., taints the same value) with a different predecessor.

Let a taint abstraction be defined by a tuple of tainted value, predecessor, and neighbor set: $a = \langle v, p, n \rangle \in A$ with $p \in A, n \subseteq A$. The result of the taint propagation is a directed taint graph $G = \langle N, E \rangle, N \subseteq A, E \subseteq A \times A$. The nodes $n \in A$ of the graph are the taint abstractions generated by the flow functions. The edges $e = \{n, m\} \in E$ describe predecessor relationships. Formally, $n, m \in E \iff n = pred(m) \vee (\exists o \in E : o \in nb(m) \wedge n = pred(o))$. Note that there is not only an edge from `n` to `m` if `n` is the predecessor of `m`, but also if `m` has a neighbor that has `n` as a predecessor. Abstractions at source statements do not have predecessors and thus no incoming edges in the graph either. Figure 10(b) shows the taint graph of the example code from Listing 10(a). Call edges are printed in red, return edges are printed in blue to improve legibility. Since the taint graph edges are constructed from predecessor relationships, the edges point in the inverse direction of the original taint propagation (“backwards”). Note that in

the taint graph, there are no more method boundaries. Since every abstraction optionally stores the statement at which it was created, one can reconstruct this information if needed, though.
After the taint propagation has finished, FlowDroid collects those taint abstractions that have arrived at sink statements. From these abstractions, FlowDroid must then traverse the graph backwards to find the respective source nodes, i.e., the head nodes of the graph from which there is a path to the detected sink abstraction. For this traversal, FlowDroid provides an interface `IAbstractionPathBuilder` through which various search strategies can be integrated into the tool. In the remainder of this Section, we describe different path reconstruction algorithms and the tradeoffs between them.
Note that the path reconstruction (i.e., the traversal through the taint propagation graph) can only be conducted after all taint propagation is complete, and not directly when a single abstraction arrives at a sink statement. In the latter case, more neighbors might be added along the path between source and sink later on, that must also be taken into account to obtain a full list sources that can influence the current sink[23].

4.13.1 Path Builder Interface

The path builder implementation is given a set of pairs $\langle a, s \rangle, a \in A$ where `s` is a sink statement and `a` is an abstraction that has arrived at that sink statement. The goal of path builder is to identify the set of sources from which, according to the taint propagation graph, the taint `a` could have originated. Optionally, a path builder can also support enumerating the path between source and sink. In this case, it also outputs the list of statements that processed the sensitive data before it reached the sink. A path builder is usually required to be sound, i.e., to never miss a possible source for a given sink. Depending on the concrete implementation, the set of sources for a given sink can, however, be a conservative over-approximation.
In the taint propagation graph, there may be more than one path between a given source and a given sink. When only finding sources (and not enumerating propagation paths), this can be ignored. Reporting the same source-to-sink connection multiple times does not add any additional value to the output for the human analyst. When enumerating the propagation paths as well, the question becomes, however, more complex. Reporting all possible paths between source and sink can lead to an exponential number of paths. If there are n branching statements in the relevant portions of the target program's code, the total number of possible taint propagation paths is in $O(2^n)$. Enumerating all of them is a prohibitive computational effort and floods the human analyst with an unusable number of reports. Therefore, we chose to not make this full enumeration the default in FlowDroid, though it can be explicitly enabled if needed. By default, if the user choses to enumerate paths, the tool only enumerates a single *witness*, i.e., one path between source and sink. This witness is selected randomly (more precisely: the tool aborts after the first path for a given pair of source and sink is found, whichever it is). This witness usually gives the analyst a good impression of the data flow for further manual investigation without overloading him or his machine. Still, it has one important drawback: If the human analyst looks at the witness to check whether a reported leak is a false positive or not, he cannot generalize his finding beyond the current witness. Marking the current witness as a false positive does not necessarily mean that there is no correct propagation path between the current source and sink. In other words, even if the witness is wrong, the leak as such might be correct. In practice, we, however, find that if the witness is a false positive, the other paths are usually similar and the complete leak can be ignored. We also exploit this observation in our work on Tasman [12] in which we use constraint solving on the FlowDroid results in order to prune false positives.
To support the path builders, all abstractions store the statements at which they were derived. If a new taint on variable `str` is derived at a statement `str = x;`, this statement will be associated with the taint. Without this information, no taint propagation path could be reported. Additionally, to allow for context-sensitive path builders, we also store the corresponding call site for each abstraction derived at a return site. Note that the current statement and the corresponding call site do not influence the equality relation of taint abstractions. This is similar to the predecessor link which is ignored in equality checks as well.

4.13.2 Context-Insensitive Source Finder

The goal of the context-insensitive source finder (CISF) is to quickly over-approximate the set of possible sources connected to a given sink. It does not support reconstructing taint propagation paths. The algorithm starts at the abstraction that has reached the sink and conducts a backwards breadth-first search. Every visited taint abstraction

[23] Incremental path building is in fact possible with additional bookkeeping. The analysis must keep a set of taint abstractions for which new neighbors have been added, after the respective abstraction has already been traversed in path reconstruction. For these new neighbors, the path reconstruction must then be triggered again.

Algorithm 2 Context-Insensitive Path Builder (CIPB) Algorithm

Require: Tainted access path ap at sink s
Ensure: Set of paths ps from source to sink

```
 1: ap.paths = ← {Path_empty}
 2: ps ← ∅
 3: Q ← [ a ]
 4: while Q ≠ empty do
 5:   curAbs ← Q.pop()
 6:   if curAbs.pred = null then
 7:     ps ← ps ∪ curAbs.paths{We have arrived at a source}
 8:   else
 9:     for each Path p ∈ curAbs.paths do
10:       curAbs.predecessor.paths p ← curAbs.predecessor.paths ∪ { p.extend(curAbs) }
11:     Q.push(curAbs.predecessor)
12:     for each Abstraction nb ∈ curAbs.neighbors do
13:       for each Path p ∈ nb.paths do
14:         nb.paths p ← curAbs.predecessor.paths ∪ { p.extend(nb) }
15:       Q.push(nb)
```

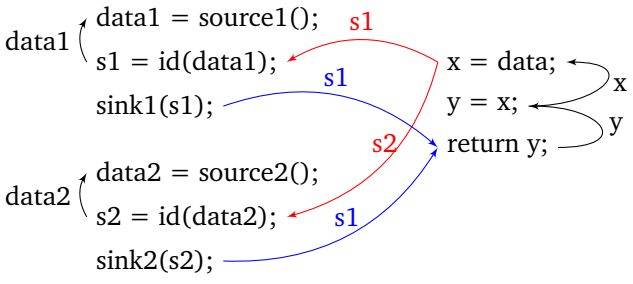
is marked as visited to avoid infinite loops. It is important to note that this path builder is multi-threaded, i.e., the searches for all sink abstractions are conducted in parallel. Keeping a set of already-visited abstractions per thread (or, more precisely: per search task) would lead to a substantial memory consumption. Instead, the abstraction objects have a field that is `null` during taint propagation and that gets filled with a bitset during path reconstruction. Every search task has a unique numeric id. When it reaches an abstraction, it checks whether the respective bit for this id is already set. If so, the task ignores the abstraction as it has already been processed. Otherwise, it looks at the abstraction's predecessors.
Note that, if multiple consecutive path reconstructions are to be performed, these bitsets must be reset. Otherwise, a new reconstruction task might arrive at a taint abstraction that is already flagged with the current id. Since the path reconstruction stores its state in the taint abstractions themselves, the reconstruction job actually alters the taint graph. During the development of FlowDroid, we accepted this non-intuitive design in favor of increased performance and reduced memory consumption.

4.13.3 Context-Insensitive Path Builder

The context-insensitive path builder (CIPB) is conceptually similar to the context-insensitive source finder. The key difference is that the CIPB also reconstructs the taint propagation path between source and sink. Consequently, the method of detecting loops on in the taint path is also different. In addition to the bitset, the taint abstraction objects also have a field for a set of *taint paths*. During taint propagation, this field is set to `null`, similar to the bitset for the CISF. When the path reconstruction is initialized, an empty path object is added to the sink abstractions. Afterwards, a breadth-first backwards search on the taint graph is performed. Whenever the path reconstruction processes an abstraction, it takes all existing paths in the current abstraction and adds the abstraction's predecessor to them. These new paths are then added to the predecessor's set of paths. If at least one path was added, the predecessor is scheduled for processing. The path reconstruction terminates if no new paths are created anymore, i.e., all predecessors have reached a fixed point on their path set. Note that the paths are made of abstractions, not statements. This makes sure to retain the maximum information as the statements are stored inside the abstractions. Algorithm 2 shows the path reconstruction in detail. It takes a taint abstraction `ap` that has arrived at a sink statement `s`. First, it associates this incoming abstraction `ap` with a set of taint paths that only contains the empty path in line 1. This path will then gradually be extended when walking up the predecessor chain. When this walk arrives at a source (i.e., an abstraction that has no more predecessors, see line 6), it will store the final propagation paths in the set `ps` as shown in line 7. For abstractions that are not sources, the path reconstruction must extend all paths of the current abstraction with the predecessor to build the path upwards one more step as shown in lines 9 and 10. A similar processing happens for all neighbors of the current abstraction as shown in lines 12 to 14.
If the target program contains loops, the taint graph can be circular. Therefore, the graph encodes an infinite number of data flow paths through the program. While picking only a single witness solves this issue, one also needs to consider that an individual path can also grow infinitely long. Keep in mind that reconstructing paths

```
1  String id(String data) {
2    String x = data;
3    String y = x;
4    return y;
5  }
6
7  void onCreate() {
8    String data1 = source1();
9    String s1 = id(data1);
10   sink1(s1);
11
12   String data2 = source2();
13   String s2 = id(data2);
14   sink2(s1);
15 }
```

(a) Path Reconstruction Example



(b) Taint Graph for Example

Figure 11: Path Reconstruction - Example as Code and Taint Graph

in the taint graph is an inherently path-sensitive problem and can be thought of as "unrolling the graph". The analysis has no upper bound on the number of times a circular subgraph (which commonly corresponds to a loop or recursion in the client program[24]) needs to be traversed. We therefoere artifically set such an upper bound and only unroll loops once. Whenever the path reconstruction algorithm detects that the current abstraction has already been added to the path, it does not extend the taint path anymore and aborts the graph traversal on this path in the graph. In the algorithm, this is implemented inside the `extend()` method. It checks whether the current abstraction with which the path is to be extended is already on the path. If so, the unmodified existing path is returned. We introduce an additional check not to push the predecessor or neighbor onto the worklist if it did not contribute any new paths.
Note that path building is always optional. Users can also select to only find the source-to-sink connections without enumerating the abstractions in between. In this case, the path that gets propagated by the ICPB only contains the latest abstraction. All previous abstractions are discarded. Infinitely growing paths are impossible in this case as the path length is always 1. Therefore, no special checking or limiting is required.

4.13.4 Context-Sensitive Path Builder

The context-sensitive path builder (CSPB) is similar to the context-insensitive path builder (CIPB) in its general algorithm. Recall that path reconstruction is done backwards (i.e., in the opposite direction of the original taint propagation), starting at the sink statement. It will thus also traverse methods in the inverse direction, i.e., from exit node to start node. When the algorithm arrives at a method call, it must find the corresponding call sides to continue the path traversal inside the correct caller. Otherwise, context-sensitivity would be lost. In Listing 11(a), the `id` function is called twice. Only the data from `source1()` is passed to `sink1` and only the data from `source2()` is passed to `sink2`. Without being able to identify the correct call site, the path reconstruction algorithm would merge these flows when returning backwards from the `id()` method, leading to false positives: a flow from `source1()` to `sink2` and a flow from `source2()` to `sink1`.
The abstraction at the start node, however, has a predecessor and usually several neighbors, one per call site. In the taint graph in Figure 11(b), there are two outgoing edges from the beginning of method `id`. With only this local predecessor information, it is not possible to distinguish the predecessors and continue only with that single predecessor that comes from the correct call site. Therefore, additional bookkeeping in the taint abstractions is required. When the taint propagation engine processes a method return, it adds a reference to the corresponding call site (the call site to which the taint propagation returns) to the taint abstraction. With this data, the context-sensitive path reconstructor can maintain a call stack as a part of the current taint path. When walking backwards through the taint graph, it checks whether the current taint abstraction belongs to a method return. If so, the corresponding

[24] When using FlowDroid's context-sensitive alias strategy, circular subgraphs of the taint graph can also be created when passing taint abstractions between the forward taint analysis and the backward alias analysis.

Algorithm 3 Context-Sensitive Path Extension

Require: Taint propagation path tp and abstraction abs
Ensure: New taint propagation path $p2$

```
1: if abs.correspondingCallSite ≠ null then
2:     tp.callStack.push(abs.correspondingCallSite){Extend the call stack}
3: if isCallSite(abs.currentStmt) then
4:     tcs ← tp.callStack.pop()
5:     if tcs ≠ abs.correspondingCallSite then
6:         return NULL{This call sequence is impossible}
7: p2 = tp.clone()
8: p2.path.push(abs)
9: return p2
```

call site is pushed on the call stack. On a method call, only the predecessor abstraction that comes from the method that is on top of the call stack is accepted. All other predecessors (current + neighbor predecessors) are discarded. Method calls can easily be detected using the current statement that is stored together with each taint abstraction. We implement this feature by further extending the `extend()` function introduced in Algorithm 2. The new function has to maintain the call stack and prune impossible paths that do not correspond to valid call-return pairs. Algorithm 3 shows the implementation in detail. Recall that each taint abstraction stores the current statement. Those abstractions that were created by return flow functions also contain a reference to the corresponding call site. In line 1, we therefore check whether the current abstraction contains a corresponding call site. If so, it was constructed at a method return. When reconstructing the taint propagation path, the algorithm traverses the taint graph in the opposite direction to the one in which it was originally constructed. At a method return, the reconstruction therefore enters a method and need to put it on the call stack (line 2). We do not speak of forward or backward here, because FlowDroid uses two solvers with different propagation directions. The normal taint propagation can already change directions when aliasing is involved as explained in Section 4.8.4. The path reconstruction traverses the resulting taint graph in the opposite direction, which means that the direction of each edge (be it forward or backward) is reversed. We therefore generalize the call stack to only speak of method calls and returns, regardless of the original direction.
When the path reconstruction leaves a method (which is at those positions at which the taint propagation originally entered it, i.e., at statements that are call sites), the algorithm must check whether the call site at the top of the call stack matches the call site which it is about to extend the path with (lines 3 to 6). This check makes sure that the mismatch displayed in Figure 11 cannot happen. When returning to the wrong call site, this statement does not match the corresponding call sie that was put on the stack when the method return was processed. Therefore, the infeasible path will be discarded.
Note that the current statement must be stored in the taint abstraction anyway to be able to reconstruct taint propagation paths as this information is not available from anywhere else. The only additional data required in the taint abstractions is the *corresponding call site* which is only non-null for abstractions created on method return edges. Conceptually, the corresponding call site for a taint abstraction is similar to the `incoming` set of the IFDS solver which links abstractions in callees to the contexts under which they were created. Theoretically, FlowDroid could, instead of matching the corresponding call sites, also track back the contexts via the `incoming` sets. This approach, however, is highly complex if multiple IFDS solvers interact and inject taint abstractions into each other's worklists which is the case with FlowDroid's flow-sensitive alias analysis. As explained in Section 4.8.4, the alias analysis' backwards solver is a second IFDS solver that is triggered during the normal forward propagation and that, in turn, can again trigger the normal forward solver. These external triggers happen at heap assignments inside methods (i.e. not at method boundaries), making it hard to reconstruct which `incoming` set to use during the backwards reconstruction. Therefore, we opted to abstract away from contexts in terms of taint abstractions and to compare call sites instead.

4.13.5 Recursive Path Builder

All path reconstruction algorithms presented so far are multi-threaded and based on a worklist. FlowDroid also supports a simpler variant of the context-sensitive path builder which is single-threaded and based on recursion instead of a worklist. This algorithm mainly exists for educational and demonstration purposes because its implementation is very concise. For larger graphs, the algorithm, however, quickly exhausts the available stack space due

to the many stack frames that need to be stored. Even if the stack space is increased sufficiently, storing as many stack frames as there are nodes on a path is highly inefficient in terms of memory consumption. On small examples, the recursive path builder can be used for cross-checking the results of the other path builders. In other words, it can save as a test oracle for the more efficient algorithms.

4.14 Code Optimization

The performance and precision of the static data flow analysis greatly depends on the code to be analyzed. In the code in Listing 37, the logging statement in Line 11 is only executed if the debug flag in Line 1 is set to `true`. Such code is common if developers want to be able to quickly switch between a development / debugging build and a production build. One can think of such constructs as equivalents to C-style preprocessor macros (`ifdef`). The difference between a preprocessor directive in C and a Java conditional as shown in the example, however, is that Java compilers usually do not erase unreachable code. The compiler output thus still contain the conditional and both of its branches. Therefore, a static analysis must assume that both branches are possible and propagate taints along both of them. In the example, this would lead to a false positive, because the debug flag is not set in this build and, consequently, the data is never leaked into the log. Aside from the loss in precision, such constructs can also lead to unecessary computation efforts. If certain additional checking and validation code is only executed if the debug flag is set, analyzing this code is unnecessary and a waste of computation time if the flag is not set. To overcome these issues, FlowDroid implements certain code optimization techniques before conducting the taint analysis. Note that for optimizations geared towards reducing the code size for faster analysis, there is always a tradeoff between the time required for applying the optimization and the time saved during the taint analysis.

```
1  private final static boolean DEBUG_VERSION = false;
2
3  boolean isDebugVersion() {
4    boolean isDbg = DEBUG_VERSION;
5    return isDbg;
6  }
7
8  void onCreate() {
9    String data = source();
10   if (isDebugVersion())
11     log(data);
12   display(data);
13 }
```

Listing 37: Interprocedural Optimization Example

The problem shown in Listing 37 can be solved by performing an inter-procedural constant value (ICVP) propagation on the whole program to be analyzed. ICVP was originally proposed by Callahan et al. [27] with a prototype implementation for Fortran and has subsequently been applied to many programming languages. The implementation in FlowDroid was developed independently from this work with the goal of (a) re-using existing transformers available in Soot and (b) achieving high performance, potentially compromising on completeness. In other words, the FlowDroid implementation rather misses a possibility for simplification than spending too much time on it.
In the remainder of this section, we will explain FlowDroid's optimizer implementation, which is shown in Algorithm 4, using the example optimization problem from Listing 37. The optimizer maintains a worklist containing those methods for which the optimization still needs to be performed. This worklist is initialized with all reachable methods in the program under analysis in Line 1 of the algorithm. For each method, the FlowDroid optimizer first applies Soot's intra-procedural constant value propagator and folder in Line 5. This step may render some assignments unnecessary , because all uses of the respective target variables have been replaced by constants. Therefore, FlowDroid removes dead code in Line 6 of the algorithm. In the example from Listing 37, these first steps propagate the value from the assignment to variable `isDbg` in Line 4 and insert it directly into the `return` statement in Line 5. This renders the original assignment (and variable) unnecessary, because all uses of this variable have been replaced with the corresponding constant value. Consequently, the assignment is removed. After these two steps, the method `isDebugVersion` in Listing 37 returns the constant value `false` and does not contain any other code anymore.
Afterwards, the interprocedural part of the analysis is conducted. It consists of two parts: (1) Propagating constants from call sites into callees (algorithm lines 7 to 12), and (2) propagating constant return values from callers back

Algorithm 4 Inter-Procedural Constant Value Propagation

Require: Set of methods *methods* in the program
Ensure: Optimized set of methods

```
 1: Q ← methods
 2: while Q ≠ empty do
 3:   m ← Q.pop()
 4:   if ¬isExcluded(m) then
 5:     intraproceduralPropagation(m)
 6:     deadCodeElimination(m)
 7:     for Parameter p ∈ paramsOf(m) do
 8:       val ← ⊥
 9:       for CallSite c ∈ callSitesOf(m) do
10:         val ← val ∨ arg(p, c)
11:       if val ≠ ⊤∧ val ≠ ⊥ then
12:         replaceUses(p, val)
13:         Q.push(m)
14:     retVal ← ⊥
15:     for Stmt rs ∈ exitStmts(m) do
16:       retVal ← retVal ∨ rs.value
17:     if retVal ≠ ⊤∧ retVal ≠ ⊥ then
18:       for CallSite c ∈ callSitesOf(m) do
19:         replaceAssignment(c, retVal)
20:         Q.push(methodOf(c))
```

into callees (algorithm lines 14 to 20). For the first direction, FlowDroid iterates over all parameters of the current method and collects all call sites of the current method. Constant propagation for a parameter is only possible if all call sites that can potentially call the current method agree on a single value for the respective call argument. If a method is called at two different call sites with different values for the same parameter, the method must stay generic to cater for both cases, and no constant can be propagated. For the same rationale, constant propagation cannot be performed if the call argument is not a constant in at least one call site. In that case, FlowDroid cannot assume that all call sites agree on the same constant call argument at runtime and conservatively refrains from concretizing the callee. In the algorithm, the lattice join in line 10 must lead to a concrete value, not the top symbol. The value is initialized with the bottom symbol. Joining the bottom symbol with a value yields that value. Joining it with the same value again does not change anything. Joining a value with a different value yields the top symbol. Therefore, the algorithm checks in Line 11 whether the lattice value is neither top (call arguments did not agree), nor bottom (no call argument found at all). If this condition is satisfied, i.e. constant propagation is possible, all accesses to the parameter inside the callee are replaced with the shared constant value obtained from the call sites as shown in Line 12. Note that this intra-procedural propagation may again lead to a constant arriving in a call site inside the callee, allowing for further inter-procedural propagation. Therefore, whenever a constant propagation across a method boundary was possible, the target method is again put on the worklist to be processed again. This also makes sure that Soot's intra-procedural constant value propagation and folding is triggered inside the callee to propagate this new constant onwards inside the callee's method body.
The second direction (propagating constant return values from the callee back into the caller) is more complex. Whenever a constant value arrives at a return site (either in the original code or as a result of the intra-procedural propagation), all calls to that method can be replaced with the respective constant value. This replacement is, however, only sound if the respective call site only has a single possible callee. Otherwise, all callees must agree on the same constant return value. Furthermore, the callee must not have any side-effects. Otherwise, replacing the call with the constant value would remove these side-effects from the program. In the example in Listing 37, both conditions are satisfied. Therefore, the constant return value from Line 5 can be propagated into the caller at Line 10. This statement now becomes `if (false)`. The algorithm iterates over all `return` statements in the current method in Line 15 and performs the same kind of lattice join operation that was already conducted for the parameter values. If the outcome is neither the top symbol nor the bottom symbol (Line 17), the algorithm iterates over all call sites of the current method. These call sites, as long as they are assignments (and not just ignore the current method's return value) are replaced by constant assignments. Afterwards, the algorithm places each

call site into the worklist. This ensures that Soot's intra-procedural constant propagator and folder is run on these methods, which can lead to further opportunities for code optimization. In the examples, this extra step removes the conditional in Line 10, because it can never evaluate to `true`. Dead code elimination finally removes Line 11.
There are some further checks that need to be conducted before propagating constants across method boundaries which we left out of Algorithm 4 for not cluttering the presentation. FlowDroid's support for explicit library models must, for instance, also be taken into account when deciding whether an assignment containing a method invocation can safely be replaced by a constant or not. A taint wrapper is allowed to also generate taints on fields as the result of a call to a library method. If the optimization would remove a call to such a method and replace it by its constant return value, the taint wrapper would no longer be able to apply its model. Therefore, as a conservative approximation, FlowDroid's code optimization checks whether the taint wrapper (if any is registered) supports the current method (see Section 4.9 for details on taint wrappers). If so, no constant propagation is performed and the call site is retained as-is. Similarly, the replacement also breaks the semantics of the program if the callee in turn calls a sink method. Replacing the call would then remove a leaking control flow path. We therefore keep method calls if they (directly or transitively) call sinks. If the callee is a source method, calls to it may not be replaced with its constant return value, either. Programs that use sensor data (such as smartphone apps) are often developed against stubs of the libraries that provide the sensor data. Though the stub may return constant placeholder data for a source that reads sensor data, the actual implementation will not. Therefore, propagating this stub value onwards would break the semantics of the target program. As a rule, we therefore exclude sources from the constant propagation. Finally, the dummy main method generated by the entry point creator (see Section 4.15) must be excluded from the optimzation. Otherwise, the opaque predicates would be evaluated, thereby damaging the lifecycle model that was originally encoded.
If the callee can potentially throw exceptions, these exceptions correspond to additional edges in the interprocedural control flow graph. If the optimization replaces a call site to such a method with a constant values, these edges get removed as well. A constant assignment cannot throw any exceptions and thus will never divert the control flow as the original callee did. To make up for these lost edges, FlowDroid creates calls to artificial static exception methods right after the constant assignment that replaces the original call site. There is one such method per exception. It uses opaque predicates to, in one branch, throw the exception, and, in the other branch, return without doing anything. This simulates that the original callee might have thrown the exception, but not necessarily. Note that these exception methods are shared between all call sites handled by FlowDroid's optimizer. Only the decision which methods to call is done per call site for precisely capturing the precise set of possible exceptions for that particular call site.

4.15 Entry Point Creation

By default, FlowDroid uses the Soot's SPARK callgraph algorithm [84] to obtain maximum precision. SPARK is based on first identifying allocation sites and then propagating this type information along assignments. As this gives SPARK a precise set of possible types for every variable (including the base objects of virtual method calls), it can accuratrely resolve call targets. This approach, on the other hand, requires that all allocation sites are visible to the analysis. This limits the usable entry points to static methods. If one wants to analyze an instance method of a class, there needs to be an allocation site that properly creates an instance of the respective class. If this allocation site is missing and the analysis starts directly inside the instance method, there is no type information for the `this` local inside the method. Consequently, no outgoing edges for any calls to `this.m()` for any method `m` can be created. The callgraph becomes not only unsound, but unusable. The new *library mode* for SPARK [114] allows more conservative approximations such as assuming any possible subtype of the class containing the instance method (CHA-style handling) when handling accesses to the `this` local. While this solves the issue of the missing edges, it can still cause a loss in precision if the exact type is in fact known to FlowDroid due to external information. Furthermore, just considering a (potentially ordered) list of methods as entry points is often imprecise. Many platforms such as Android or the Java Applet Engine define a lifecycle for the user code, i.e., complex rules on when and whether certain user code methods are called. To capture these semantics, all of these rules must be modeled. Just assuming that every method can be called at any time will likely lead to an over-approximation and thus false positives.
FlowDroid therefore provides an interface for *entry point creators*. An entry point creator generates a dummy main method which serves as an artificial entry point for callgraph construction. This method can instantiate the correct target classes to be analyzed and call the desired instance methods in the right order. Conceptually, an analysis of some instance method inside a larger program is reduced to analyzing a normal Java program. The dummy main method takes the place of Java's normal `main()` method for the purpose of the analysis. Note that the resulting

fake program is not required to compute any meaningful output when executed by the JVM. It might even crash with an exception or violate requirements of the Java specification. The dummy main method is only required to be equivalent to the actual use of the target methods with respect to the static analysis (consisting of the callgraph construction algorithm and the FlowDroid static data flow tracker). From this dummy main method, the callgraph algorithm can derive precise type information and thus generate a precise callgraph.
The dummy main method can also be used to emulate external platform behavior in a flow-sensitive analysis. When analyzing Android apps, Java Server Pages, or Java Applets, the target code is a plug-in into a larger system. A Java applet is implemented by creating a new class that inherits `java.lang.Applet`. The framework (the applet viewer or the JVM plugin inside the browser) is responsible for instantiating this user-defined class and for calling certain methods such as `init()` to execute the user code. Note that these method invoked by the framework are ordered. The call to `invoke()` will always happen before the call to `destroy()`. A flow-sensitive analysis must take this order into account. The easiest approach to faithfully handle such constraints is to model the framework behavior in the dummy main method. In this case, the dummy main method is not only the seed for callgraph construction, but also provides those parts of the control flow to the analysis that would otherwise be hidden in the (usually non-analyzable) library code. This is especially important in the context of Android as we will show in Section 5. For the common use cases of the data flow analysis FlowDroid provides the following entry point creators:

- **SequentialEntryPointCreator** The dummy main method generated by this class simply contains a sequential list of calls to the target methods.
- **DefaultEntryPointCreator** This class generates a dummy main method that simulates that every target method is called an arbitrary number of times in an arbitrary order. Therefore, this class can be seen as a conservative over-approximation of arbitrary lifecycles.

When generating a dummy main method, several challenges arise. In the remainder of this Section, we will discuss these challenges. In FlowDroid, these aspects are centralized in the abstract base class `BaseEntryPointCreator` from which all concrete entry point creators are derived.

4.15.1 Creating Class Instances

If one wants to analyze an instance method `m()` in a program or a library, the dummy main method must first create an instance of the parent class and then invoke the method `m()`. This means that it must first call a constructor of that class. For simplicty, we assume that all constructors are semantically equivalent, i.e, it does not matter for the call to `m()` which constructor was initially used to create an instance of the parent class. This assumption is in line with good programming practice and class design, but can of course be exploited by a determined programmer who wants to hide properties of his program from the analysis. Based on this assumption, the entry point creator can collect an ordered list of public *candidate constructors*. The order of constructors in the list reflects the effort for calling the respective constructor. The effort is directly proportional to the number of arguments. For every parameter, the entry point creator must find a suitable (fake) value. Therefore, a call to a constructor that requires no parameters is easier to model than a call with one parameter, which is in turn easier to model than a call to a constructor with a dozen parameters. Note that simply passing `null` as a parameter whenever an object type is required does not faithfully model the semantics of many constructors. Recall that the SPARK callgraph algorithm works by propagating type information from allocation sites to call sites. If a class saves a constructor parameter's value into a field and later calls a method on that value, passing a `null` value to the constructor will lead to missing callgraph edges. If two constructors have the same number of arguments, we pick the one with the larger number of primitive parameter types.
Once the list of candidate constructors has been created, the entry point creator tries to create calls to these constructors, starting with the simplest one. If this fails, for instance because no suitable parameter values could be found, the next candidate constructor in the list is taken. Finding a value for each parameter is done by a recursive algorithm. Primitive types are the base case for the recursion. For them, hard-coded default values can be used. Note that we also consider strings as primitives as they are immutable. For object types, the algorithm recursively tries to construct an instance (i.e., a constructor call in the dummy main method) of the respective class. If the constructors of this class again take object types as parameters, the recursion continues by constructing instances of those classes as well. The recursion stops if it either reaches a simple constructor call (i.e., that takes no parameters or only primitive-typed ones) or if the algorithm runs into a loop. In the latter case, the algorithm uses `null` as a parameter for the constructor call that has caused the instance creation loop. Again, the idea is to model

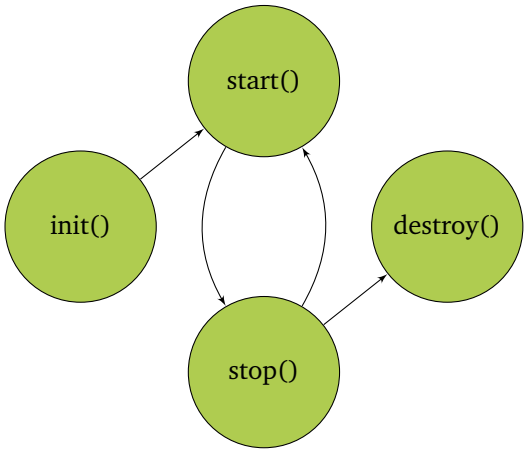
the target class use as faithfully as possible. Therefore, the `null` value is not injected into the top-level constructor call, but at the position at which the loop occurs.
This model so far assumes that whenever a constructor is invoked, a new class instance is created for each parameter that is of an object type. While this is technically correct, it fails to capture relationships between separate constructor invocations. Note that usually more than one target method must be called by a dummy main method, which in turn, usually requires invoking the constructors of multiple classes. Consider the case of an inner class. In Java, the constructors of inner classes always take an instance of the respective outer class as a parameter. If the construction of outer and inner class are considered separately, a new instance of the outer class is created when creating the instance of the inner class, even if the outer class already exists because of an earlier method call. If the inner class accesses fields from the outer class, this leads to lost taints in a field-sensitive analysis as the base objects do not match anymore. The analysis would capture that the two instances of the outer class are separate and do not alias.
Since the use of inner classes (and other dependencies between classes) is common in Java programs, the entry point creator must avoid such discrepancies. In FlowDroid, we use the following heuristic: Whenever a constructor call is generated, the entry point creator saves the target variable of the new instance in a map from type to variable. If a parameter of that type is needed later on, the entry point creator performs a lookup in this map to see if there is already an instance. If so, the variable of this instance is passed as a parameter. Only if no instance exists yet, a constructor call is generated. This maximizes instance re-use. In some cases, it may lead to unnecessary connections between classes, but since the process is fully automated, no information about the actual use of the target classes is available. Thus, one needs to make assumptions and provide heuristics that fit most (or at least the most important) cases. In our tests, this heuristic yields correct results.

4.15.2 Creating Method Invocations

Method invocations are similar to constructor invocations. For instance methods, the entry point creator first needs to create an instance of the respective class as described above. Finding parameter values for the method also works exactly as for constructor calls. The only difference is that the entry point creator needs to perform a virtual method lookup. If it shall call a method `m` in class `c`, but class `c` inherits that method unchanged from its superclass `d`, then the call target must reference `d.m` as `c.m` does not exist. Requiring the input specification to already take care of virtual method lookup has proven to be impractical. Methods can move from a class to its superclass when libraries or frameworks change, even in minor updates. Allowing for such inaccuracies in the specifications makes the entry point creator gracefully handle such changes.

4.15.3 Modeling Call Sequences

When capturing the interaction between the framework and the application under analysis in a dummy main method, control flow is not necessarily sequential. A Java applet, for instance, can be started and stopped by the runtime at any time. After stopping a component, it can decide whether to destroy the component completely or whether to restart it and loop through its lifecycle once again, see Figure 12(a). The entry point creator must account for such non-linear behavior. A static analysis cannot a-priori decide which path through the lifecycle the framework will use, so it must account for all possible paths. In other words, it must encode a non-deterministic finite state machine (NFA) in which the transition function corresponds to the lifecycle methods called by the framework. A simplistic approach would enumerate all paths through the NFA similar to loop unrolling in symbolic execution, i.e., generate a set of purely sequential lifecycle simulations. This approach, however, inherits the bounding problem of loop unrolling. In general, the number of iterations through the lifecycle that the framework will conduct at runtime is unknoiwn to the static analysis and potentially unlimited. Therefore, the analysis would need to introduce an artificial upper bound, making the analysis unsound. Furthermore, the set of emulated lifecycle instances (i.e., paths through the NFA) can grow very large, imposing scalability issues on the analysis. Consequently, unrolling is not a solution for modeling non-linear lifecyles.
Instead, non-linear call sequences are modeled using *opaque predicates*. At every point where the control flow can divert, the entry point creator inserts a conditional as shown in Listing 12(b). Note that the listing uses a bytecode-like notation that allows labels and jumps. The predicate of the conditional is designed such that it cannot be evaluated by the static analysis (`opaque()` in the example). The analysis must thus assume that both outcomes are possible and analyze both the `then` branch and the `else` branch as possible successors of the conditional. Semantically, this exactly matches a non-sequential lifecyle. The analysis must propagate all facts it has dervied so far into both branches and continue the analysis independently for each branch. Note that this approach does



(a) Java Applet Lifecycle

```
1  static void dummyMain(String[] args) {
2    l0: MyApplet applet = new MyApplet();
3    applet.init();
4    l1 : applet.start();
5    applet.stop();
6    if (opaque())
7      goto l1;
8    applet.destroy();
9    goto l0;
10 }
```

(b) Java Applet Entry Point Code

Figure 12: Java Applet Lifecycle Schematic and Entry Point Pseudocode

not suffer from the path explosion problem that arises when enumerating all paths, since the analysis can merge data flow facts on merge points. It does not compute one solution per possible path, but a single solution that is a merge of all possible paths through the lifecycle. This exploits that IFDS, on which FlowDroid is based, computes a *meet-over-all-paths* solution.

The design decisions made when generating the dummy main method significantly influence the precision, recall, and performance of the data flow analysis. Obviously, missing calls to lifecycle methods can lead to false negatives for the overall analysis. Some tradeoffs, on the other hand, can be subtle. If a program under analysis consists of multiple concurrent components, the analysis developer who implements the dummy main generator must decide on which granularity he needs to model the possible runtime interactions between the components. The most trivial approach is to generate a sequence of independent lifecycles, which can be thought of as multiple copies of the same template beneath each other. While this approach is simple to implement, it misses leaks that are caused by concurrent interactions. One component could, for instance, write tainted data into a static field, which is read and leaked by another component, before the first component overwrites it with `null` again. In this scenario, analyzing neither component alone (or as part of an unconnected sequence) is sufficient for detecting the leak. On the other hand, assuming that concurrent interleaving is possible after each method call, requires a more complex structure of conditions with opaque predicates. Furthermore, it can also lead to irrelevant leaks being detected, i.e., leaks that are technically possible under the given model, but that very unlikely or even impossible to occur at runtime. Note that the method level is the most fine-grained granularity of interleavings possible with our concept of a dummy main method. Already this granularity is, however, infeasible in practice. Most commonly, dummy main methods therefore treat components individually and rely on external semantic models to precisely capture the inter-component communication semantics supported by the target platform. For the case of Android, we will disucss this issue further in Section 5.6.2.

4.16 The Overall FlowDroid Workflow

The description of the FlowDroid data flow tracker has so far been focused on the various components in the architecture of the data flow tracker to simplify the presentation. In this section, we will present how the components fit together in a full workflow from input parsing to writing out the discovered data flows. As shown in Figure 13, FlowDroid first initializes a Soot instance by loading Soot's basic classes that must be available in every Soot `Scene` such as `java.lang.Object`. At this point, Soot also initializes its classpath for dependencies. Recall that FlowDroid explicitly configures Soot not to load the bodies of classes from the Java Standard Library or the Android SDK and instead leaves library handling to the taint wrapper as explained in Section 4.9. Nevertheless, the class definitions and hierarchy are loaded to make sure that type checks can be performed. This is not only important for resolving polymorphic call sites, but also for the type checking explained in Section 4.11. Also note that FlowDroid cannot call Soot's standard `main()` method, but instead needs to explicitly trigger the class loading

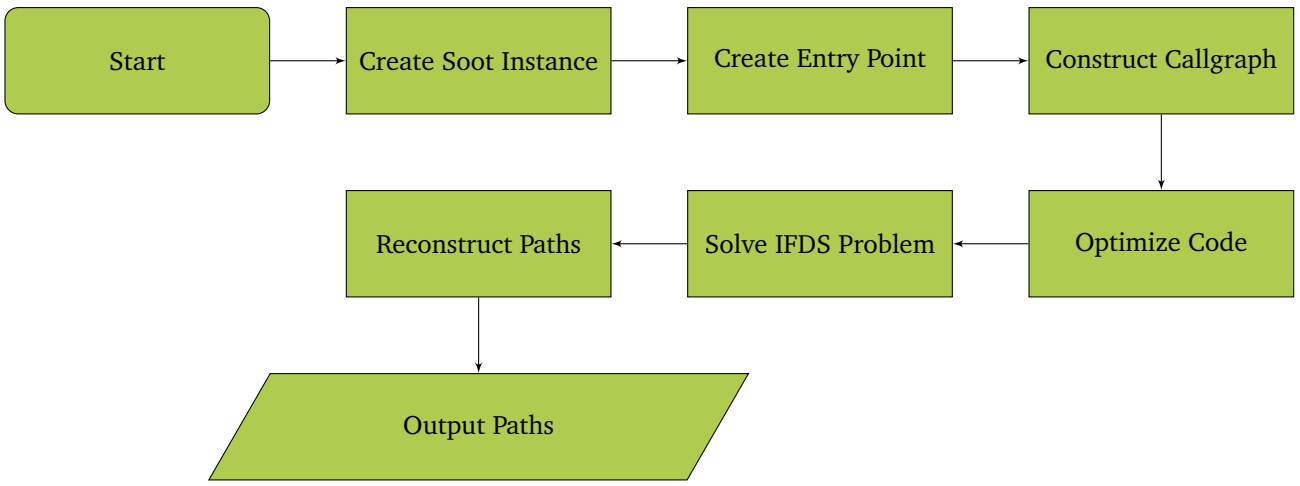


Figure 13: FlowDroid Workflow Diagram

and initialization, because Soot's `main()` method would immediately execute transformations such as callgraph construction. FlowDroid, on the other hand, must fit additional steps such as creating the entry point in between. After Soot has been initialized, the entry point is created as described in Section 4.15. This entry point emulates the `main()` method which is not available for many analysis targets. This is especially important when analyzing Java applets or Android apps that tightly integrate into their respective execution environments. Given this entry point, Soot's built-in algorithms can then generate the callgraph. Once the callgraph exists, the IFDS solver can be started on the the IFDS tabulation problem that models the FlowDroid core. Note that, depending on the alias algorithm to be used, a second IFDS solver is spawned to process the backward propagation of alias taints. All solvers keep track of the taint propagation tasks they have spawned. The IFDS step is considered to be finished when the task counter of all IFDS solvers has reached zero. Note that as long as there is still at least one active task in a single solver, this task is able to potentially spawn new tasks in any of the solvers.
Once all spawned IFDS solvers have finished, FlowDroid has a set of abstractions that have reached sink statements. From these sink abstractions, the original sources as well as the taint propagation paths are reconstructed as explained in Section 4.13. The results of this step are the final results of the FlowDroid analysis. Depending on the configuration, these data objects are either returned directly for further processing in an external analysis tool, or are serialized into an XML file on external storage.

4.17 Supported Language Constructs & Limitations

This section describes the Java language constructs supported by FlowDroid and discusses the limitations of the data flow engine with regard to Java language features. Note that this section only serves as a summary. The details of the explicitly supported features have already been discussed in the respective previoius sections.

Arrays

FlowDroid over-approximates the taint state of arrays by considering the complete arrays as tainted once a single element inside it gets tainted as explained in Section 4.4. This ensures that no leaks are missed, but can lead to false positives. On the other hand, FlowDroid is able to distinguish whether only the elements of an array, or its length, or both are tainted.

Dynamic Code Loading

FlowDroid assumes the complete program under analysis to be present when performing thr analysis. If the program under analysis loads additional code at runtime, FlowDroid ignores this load operation and does not resolve the location from which the code is loaded. To cover such code as well, a human analyst can, however, manually obtain a copy of the dynamically-loaded code and present it to FlowDroid as an additional input. Technically, FlowDroid can put an arbitrary number of code files on Soot's `process-dir` parameter and construct a common Soot `Scene` from the union of all classes in all of these files. In other words, this will perform a simplistic merge operation that assumes that there are no naming conflicts. In the Java case, one can compare this approach to mer-

ging all `class` files from all `JARs` into a single `JAR` which is then used as the analysis target. In many cases, this is an appropriate solution to the dynamic-code loading. Note that there are inherent complexities involved in further automating the merging. The location from which the additional code is loaded might not even be accessiblke to a static analysis if it is, for instance, read from a configuration file.

Exceptions

FlowDroid supports exceptions in two different ways. Firstly, the inter-procedural control flow graph contains exceptional edges, i.e., exceptions from statements that can potentially throw an exception to all possible exception handlers that are valid for the respective code range and excetion type. Secondly, if tainted data is stored inside an exception object which is then thrown, FlowDroid is able to track the data flow to the exception handler. If the exception handler reads the tainted data from the exception object and passes it to a sink, this leak is correctly detected. More information on FlowDroid's exception tracking can be found in Section 4.5.

Implicit Flows

Sensitive data cannot only be transferred through explicit assignments, but also by making the program's control flow dependent upon the value of the sensitive variable or field. FlowDroid can optionally track such control-flow dependencies and report leaks if sinks are only called under conditions dependent on sensitive data, or if the leaked data is computed in a way that is control-flow dependent on a tainted value. We refer the reader to Section 4.6 for further details.

Native Code

The Soot framework, on which FlowDroid is based, was originally designed for analyzing and transforming Java code. Consequently, the Jimple intermediate representation has no means for representing low-level constructs such as pointer operations that are prevalent in native code. Therefore, integrating native code analysis into Soot is a non-trivial undertaking. Soot abstracts from the native code at the interface level, i.e., at the Java part of the JNI declaration. In other words, methods can be marked as `native` and are then treated just like abstract methods. FlowDroid can make use of a native call handler (see Section 4.10) to provide explicit data flow models for these methods. We provide pre-defined models for common native methods such as `System.arraycopy()`.

Recursion and Loops

Since FlowDroid is a path-insensitive data flow tracker, it allows for arbitrary-length recursions. Though the program's control flow is circular, the taint state is guaranteed to reach a fixed point. The solver does not keep track of how often a variable is assigned tainted data. Once the tainted access path has a reference to all possible predecessors that assign tainted data to it (regardless of how often the respective statements are executed), it no longer needs to process that access path anymore. When no new distinct access paths and no further predecessors are added anymore, the global fixed point is reached and the analysis terminated. Refer to Section 4.13 for a detailed explanation on how FlowDroid uses predecessor references instead of propagating pointers to the respective source along with each taint abstraction.

Note that when the user chooses to reconstruct taint propagation paths, i.e., the paths that a tainted varoiable has taken through the program between source and sink, recursion and loops can become and issue. If there is a loop or a recursive method call along the path, then there is one possible path per execution count. In other words, there is one path from source to sink on which the loop was passed once, another path on which it was passed twice, etc. We solve this issue by not enumerating all the paths between source and sink, but instead returning one witness that shows the source-to-sink connection by example. This will usually be the path that either skips the loop entirely or executes it only once (in case at least one execution is necessary to create the leak), because this is the shortest path possible and will thus be reconstructed first.

Reflection

FlowDroid inherits its support for reflective method calls from the Soot framework on which it is based. Consequently, it can handle all those reflective call sites for which the SPARK callgraph algorithm integrated into Soot can find outgoing call edges. SPARK is based on propagating type information from allocation sites to virtual call sites. In the case of reflective calls, this type information is, for instance, available, if the base object of the call is constructed as usual (i.e., without using reflection), and only the virtual method call itself is done using reflection. Additionally, SPARK also automatically converts calls to `class.forName()` with constant arguments (i.e., class names) into class constants. It creates fake allocation sites when the `newInstance()` method of such a `java.lang.Class` object is

called. Afterwards, the type propagation towards the virtual call site works as usual, without requiring any further special-casing for reflection.
Given that type infomation for the base object of the virtual method call is available, SPARK resolves reflective method calls by matching the types of the arguments to potential receiver methods. It looks for candidates in the propagated base type and all of its subtypes. Each method that takes parameters that are cast-conmpatible with the given argument types at the call site is considered a potential callee. Note that this approach can lead to false positives if unrelated methods with the same argument types, but different names are found. At the moment, SPARK does not check any constraints on the method name, even if the name of the target method is available statically. SPARK's reflection support is work in progress (including contributions from the author of this thesis) at the time of writing and is likely to improve with regard to both precision and recall in the future.

Static Fields

Static fields can receive taints just like any other field. This is handled by the FlowDroid Core (see Section 4.2).

Threads

Threads allow code to be executed in parallel. On modern multi-core processor architectures, this leads to true parallelism, i.e., the order in which statements in different threads are executed relatively to one another is completely undefined aisde from explicit synchronization introduced by the programmer. This challenges static analysis approaches which traditionally rely on a single control flow that executes all statements of a whole program execution sequentially. When this ordering constraint is removed, new types of data- and control-flow dependencies can be introduced. Consider a scenario with two threads, in which one thread reads sensitive data from a source and writes it into a field. The second thread reads the data from the field and leaks it. In normal sequential code, there would be an order that defines whether the data is first read, then transmitted through the field and finally leaked, or whether the read happens before the write and thus no leak occurs. In a concurrent program, it depends on external effects such as scheduling and system load, which of the two scenarios happens at runtime. To be complete and not miss leaks, an analysis would have to take into account all possible thread interleavings and check whether they can lead to leaks. Modelling all interleavings, however, is a challenge on its own, because the analysis must then precisely handle synchronization constructs and atomic operations provided by the JVM (such as the `AtomicInteger` class as opposed to a normal `int`). An imprecise thread model, on the other hand, would be less complex, but is likely to introduce a considerable number of false positives. In this work, we therefore leave the issue of thread interleaving [25] aside and assume that each thread completely executes when it is started. In other words, we inline the thread code at the position where the respective thread is started. On a conceptual level, this design decision assumes that there are no dependencies between threads at all. Threads are rather assumed to process separate atomic work items. In this model, a thread is allowed to consume data from other threads when it starts, but then needs to execute atomically. Once it has finished its work, it can pass back data to other threads. Technically, this reduces the problem of thread handling to creating a callgraph edge from `Thread.start()` to the `run()` method of the respective thread.
Threads can be created using various APIs, including ones that are present in the normal Java class library and special ones added by the Android framework. For creating the aforementioned fake edge to `Thread.run()`, all of these APIs must be modeled. FlowDroid inherits support for some basic cases from Soot's SPARK implementation. For other cases, the tool injects stub implementations of the respective API that are equivalent to the original implementation in terms of data flow, but that are minimal in terms of complexity and code size. This stub injection is explained in Section 4.9.3.

Virtual Dispatch

FlowDroid relies on existing callgraph algorithms to resolve virtual call sites. As explained in Section 4.2.6, the data flow tracker uses Soot's integrated SPARK algorithm by default. SPARK offers higher precision than the other callgraph algoroithms integrated into Soot (CHA, VTA, RTA). The core concept of SPARK is to propagate type information from allocation sites to call sites. This type information is then used to limit the possible receiver classes for the call. Still, because SPARK is context-insensitive, some false positives occur, especially with callbacks such as Java's built-in threading mechanism. Our experiments on DroidBench (see Section 7.2) show that FlowDroid, together with its own type propagation feature for tainted objects (see Section 4.11), is highly precise in practice when resolving the targets of virtual call sites.

[25] The Joana static data flow analysis supports the RLSOD criterion for concurrent programs [25]

5 Static Data Flow Analysis for Android

In the previous sections, we have provided the necessary background knowledge and presented the generic part of the FlowDroid data flow tracker. Recall that FlowDroid's data flow engine is generic with the goal to support arbitrary target platforms. The platform-specific parts of the FlowDroid static flow analysis such as modeling sources and sinks, and dealing with library methods have therefore been decoupled from the core solver using interfaces. The default implementations of these interfaces discussed in the previous section model the behavior of the Java Runtime. They allow FlowDroid to analyze traditional Java programs for data flows and information leakage. In this section, we now describe the Android-specific extensions to FlowDroid. In general, we explore Java programs and Android apps as analysis targets in this thesis. Other researchers have applied FlowDroid to web services based on Java EE.
We will first discuss how the lifecycle of Android apps is modeled in Section 5.1. Afterwards, we will expain how FlowDroid deals with references to UI ocntrols in Section 5.2, before going over the details of how we handle the omnipresent callbacks provided by the Android framework in Section 5.3. In Section 5.4, we present various optimizations for making the analysis more efficient. Section 5.5 presents related work and shows how other researchers handle the Android-specific parts of a static analysis. Finally, we present the most important extensions to FlowDroid in Section 5.6.

5.1 The Android Component Lifecycle

Android apps can be thought of as plugins into the Android operating system rather than as standalone apps. A normal Java program has a `main` method that is invoked by the runtime environment and that is then completely self-responsible for the remainder of the program execution. The Android runtime, on the other hand, is much more tightly coupled with the apps it executes. Developers create apps by implementing own Java classes that inherit predefined system classes and by overwriting callback methods in these classes. This allows the Android OS to, e.g., pause or resume the execution of an app at any time. Apps must be designed to tolerate such external influences without losing their state or corrupting any data. This tight coupling between the operating system and the apps is mainly due to the restrictions of mobile devices. Shared resources such as processing power, memory, or energy, are much scarcer than on a traditional desktop or even laptop computer. If the user switches between apps, it may thus become necessary to re-allocate these resources to the new foreground app and away from the old one. This can include pausing or even stopping the old app altogether to free up memory. When the device's battery gets low, the operating system may also stop or pause apps in favor of other apps or the system itself.
An Android app consists of a set of well-defined *components*. There are different component types for different tasks. An *activity* represents a single, focused task in the graphical user interface such as selecting a song to play. A *service* is used for long-running background tasks that are independent of user interaction such as playing the selected song, even when the user switches to a different app. A *broadcast receiver* listens for system-wide broadcast messages such as the request to mute all audio output for silent mode. A *content provider* models an application-specific database such as the media library of the music player app. For each component type, the Android SDK provides a base class from which the developer can inherit. Furthermore, each component type has its own lifecycle that defines how the operating system can interact with the component by calling methods pre-defined in the base classes. Conceptually, this retricts the possible sequences of method calls made by the operating system. Figure 14 shows the activity lifecycle[26]. The activity has the most complex lifecycle of all four component types. Still, even Figure 14, which is taken from the official Android documentation, is incomplete. As explained above, an activity must always be prepared to save its state before being stopped, and to restore its state after being resumed. For this state handling, Android provides the two additional lifecycle methods `onSaveInstanceState` and `onRestoreInstanceState`. Carefully reading the documentation (including the API's Javadoc) and investigating the Android source code reveals more such methods (e.g., `onPostCreate`) that are missing from the lifecycle graphics. Nevertheless, all these methods must be added to the lifecycle to precisely capture the actual interaction between the Android operating system and the app at runtime.
For the static analysis, the component lifecycles are modeled by creating a dummy main method as explained in Section 4.15. The Android extensions for FlowDroid therefore provide an own, specialized entry point creator implementation. This entry point creator parses the `AndroidManifest.xml` configuration file which is present in every Android app. The manifest is a declarative specifiication of the app's components. Conceptually, it registers the components with the operating system: Which components do exist, for which activities shall there be icons

[26] Picture taken from the Android documentation at `http://developer.android.com/guide/components/activities.html`

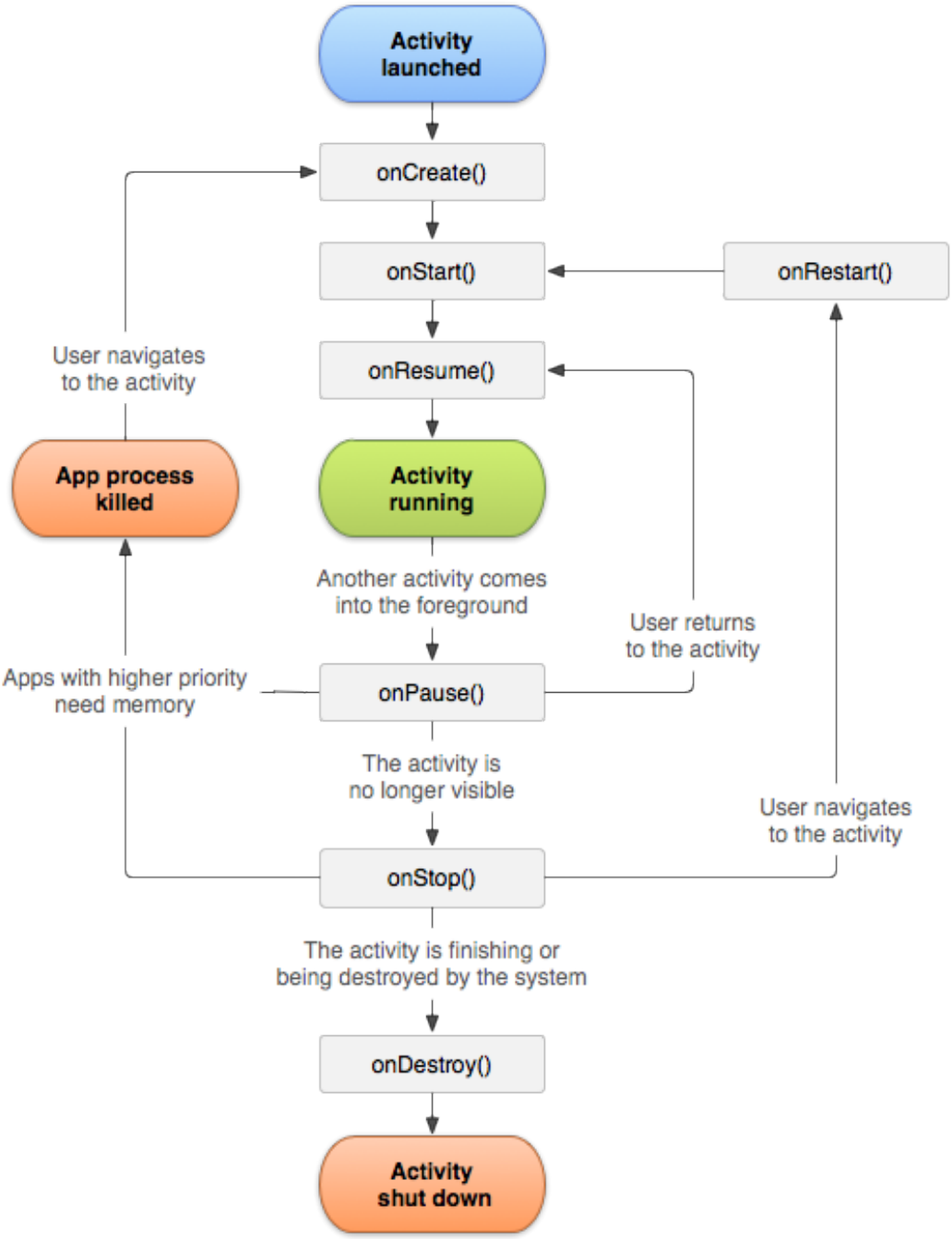


Figure 14: Android Activity Lifecycle

in the phone's launcher, etc.? For every component in the manifest, the entry point creator checks which lifecycle methods it overwrites. For each such method, a corresponding call is then created in the dummy main method. As explained earlier, the entry point creator uses opaque predicates to model optinal jumps through the lifecycle, such as after `onPause`. The activity may then either be destroyed, or resumed, leading to two possible successor callbacks after `onPause`. The opaque predicate models that both of them are possible and must be consideres valid paths through the lifecycle.
Note that we model component lifecycles independently from each other. A state transition in one component's lifecycle does not trigger or influence any state transitions in other components. Since the Android framework is single-threaded, there is only one lifecycle method active at a time. In other words, if an activity and a service are running inside the same app at the same time, the Android framework uses the same event queue for them. Whenever a lifecycle method from one component has finished, the framework can either call the nextr lifecycle method from that component, or can perform a context switch and call the next lifecycle method from the other component. There is no a-priori definition of when such context changes happen. Theoretically, this allows for hidden dependencies between components, because all components share their memory space and can, e.g., access the same static variables. Thoeretically, one component can write sensitive data into a static variable and the other one (which is running at the same time) can leak it on the next context change. Since the context changes are, however, not known a-priori, we assume that apps do not use such unreliable channels. Furthermore, since FlowDroid is a purely intra-component data flow tracker, we simply place the lifecycles of all components beneath each other and only simulate that they can be executed in an abritrary order using opaque predicates. In our model, a single component lifecycle, however, is atomic and cannot interfere with other component lifecycles.

5.2 UI Control Handling

Android apps are usually highly interactive. The Android framework contains various readily available mechanisms that app developers can use to display user interface controls and interact with them. For a data flow analysis, modeling an app's user interface is important in two ways: Firstly, user interface controls may be associated with callback handlers for e.g., button clicks or key presses inside a text field. Omitting these callbacks would lead to an incomplete callgraph and thus potential false negatives in the data flow analysis as explained in Section 5.3. Additionally, UI elements are, however, also potential sources of sensitive information. Assume that a user enters a password into a text field. If this password is leaked to an adversary, the user may be subject to identity theft or allow the attacker access to other privacy-sensitive services. Therefore, a data flow tool for Android such as FlowDroid must build a complete model of an app's user interface and look for sensitive interactions.
Not all inputs are sensitive, though. A weather app will, for instance, rightfully leak the user's city of interest to a remote web server in order to obtain the weather forecast for that city. Even if this data is also leaked to an additional third-party, it might not infringe upon the user's privacy. Therefore, FlowDroid supports three different modes for handling data entered through UI elements:

- Ignore all UI data. This leads to a potentially incomplete set of leaks.
- Only consider password fields as sensitive inputs. This is the default as it captures the implicit expectations of most users and provides a balance between incompleteness and a technically correct, but useless flood of detected leaks.
- Treat all UI input elements as sources. While this option makes the UI analysis complete, it can greatly increase the number of detected (expected) leaks. Note that this option still excludes non-mutable UI elements such as buttons as they cannot obtain any information from the user.

In Android, the user interface of an app is described in *layout XML files* as shown in Listing 38. In other words, the layout XML file is the visible part of an activity or fragment. The example shows a text input field that is configured to accept passwords, i.e., hide its contents. At compile time, the layout XML files are converted from normal plain text into Android's internal, binary XML file format. The control names are erased. Instead, they are only referenced by unique numeric identifiers which are unique across the whole app. If the developer wants to reference a UI element or layout control from his code, he can identify it through a constant defined in an automatically generated resource file called `R.java`. The names of the constants in this file (`R.id.pwField` in the example) correspond to the names the developer has given to his layout controls in the XML file via the `id` tag (`pwField` in the example). The constant value corresponds to the automatically-generated unique identifier of the layout control. For accessing a UI element, the developer calls the `findViewById` API function with this value as

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <EditText
        android:id="@+id/pwField"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="150dp"
        android:ems="10"
        android:inputType="textPassword" >
    </EditText>
</RelativeLayout>
```

Listing 38: Password Field in a UI Layout

```
public class PrivateDataLeak2 extends Activity {
    @Override
    protected void onStart() {
        super.onStart();
        EditText mEdit = (EditText)findViewById(R.id.pwField);
        Log.v("Password", mEdit.getText().toString());
    }
    ...
}
```

Listing 39: Accessing UI Elements in Code. Adapted from PrivateDataLeak2 in DroidBench.

shown in Figure 39. The returned UI model object allows read and write access to the layout properties (positions, sizes, etc.), as well as the contents of the control.
Note that Android's `dx` compiler performs a constant propagation during compilation. The classes in `R.java` are erased. Instead, the values of the automatically-generated constants are directly injected into the calls to e.g., `findViewById`. This leaves the compiled app without any semantic layout information in the code except for these IDs. FlowDroid must therefore map the IDs back to layout controls to find potential sources. Recall that IDs are unique across the whole app, even if the app contains multiple activities. For correctly resolving accesses to UI elements, one therefore does not need to associate a layout XML file with its corresponding activity. One only needs a mapping from the unique identifier in the code to a model of the corresponding UI element with the same ID. FlowDroid therefore conducts the following steps to build a UI model and collect the UI sources:

1. Parse all layout XML files. Inside an APK file (which are just renamed `zip` archives), the folder `res/layout` contains all layout XML files. For every layout control, add a mapping between ID and the UI model element.

2. Create artificial sources for all calls to `findViewById()` that are passed IDs of sensitive UI controls. Depending on the configuration explained above, this includes a check of the element type (only consider password fields or treat all UI input elements as sources). Static controls such as buttons, etc. are always excluded.

The Android layout XML format also supports indirections. One layout XML file can reference other ones using an `include` directive. These files are then merged into the same layout when the respective app is started. FlowDroid statically merges these includes into the model. A simplistic layout file parser could also ignore include directives and simply load all xml files in the layout directory. While this would be sufficient for resolving layout control IDs referenced from the source code, it does not faithfully represent the hierarchical structure of the original UI layout. This structure will, however, be important when analyzing UI callbacks in Section 5.3.
As usual in Android, references in include directives of layout XML files are created as plain strings at development time, but are converted into numeric identifiers at compile time. As the files inside the APK container still have their original name, the generated ID is not sufficient to locate the referenced file in the APK. This creates additional complexity as FlowDroid must resolve these IDs to obtain the file name of the respective target file. Such ID

```
1  void onCreate() {
2    LocationManager mgr = (LocationManager) getSystemService(Context.LOCATION_SERVICE);
3    mgr.requestLocationUpdates(LocationManager.GPS_PROVIDER, 5000, 10,
4      new LocationListener() {
5        @Override
6        public void onLocationChanged(Location loc) {
7          sink(loc.toString());
8        }
9        ...
10     }
11   );
12 }
```

Listing 40: Location Leak in Callback

references are pointers into the global resource configuration database (`resources.arsc`) included with every Android app. The resource configuration contains mappings from IDs to resource objects for potentially multiple configurations. One configuration could for instance translate ID 1001 to "Hello" and gets activated on English phones, whereas a different configuration could translate the same ID 1001 to "Bonjour" and gets activated on French phones. Similar distinctions between configurations can be made based upon screen size, screen orientation, type of available keyboard, or Android API version. A static analysis tool such as FlowDroid cannot identify which configuration will be used at runtime. One possibility would be to merge all possible configurations to ensure soundness. Since data leaks usually, however, do not depend on configuration-dependent constant value lookups in resource files, FlowDroid opts to only consider the first (and usually only) configuration by default.

5.3 Callback Handling

Smartphones and tablets nowadays not only provide computational resources, but also various types of sensors that can be read by apps. In addition to traditional readings (get the current value of the sensor), the Android operating system also provides event-driven access to this sensor data. The latter allows the app to be notified whenever the sensor value changes, e.g., the smartphone is moved to a different physical location and new GPS data is available. These events can happen at any time in an arbitrary order. The respective callbacks inside the app are then invoked by the Android operating system. Note that apps need to explicitly subscribe for receiving a particular kind of events such as location changes. Otherwise, no events are delivered, i.e., no callback is invoked. This distinguishes event-driven callbacks from the lifecycle methods described in Section 5.1. The lifecycle methods are always invoked if the respective component changes its status (regardless of any subscription) and can only occur in a pre-defined order.
For a static data flow analysis tool, handling callbacks is important for obtaining a complete callgraph. The dummy main method explained in Section 5.1 must invoke the callback methods during the simulated runtime of the component hosting the particular callback. If an activity, for instance, listens for GPS location changes, the dummy main method must simulate an arbitrary number of invocations of this callback method between the `onResume` and `onPause` lifecycle events. An activity that has not been started yet or that has been paused or stopped cannot receive events. Placing the call at the right position is important to correctly model leaks where e.g., the sensitive data is obtained in the `onCreate()` method of an activity and leaked in the callback that fires when a certain timer expires. Such time-dependent leaks are called *timing bombs* and are quite prevalent in modern malware [33].

5.3.1 Dynamic Callback Registration

An Android app can subscribe to events by calling particular pre-defined API functions. In the example in Listing 40, the app first obtains a reference to the `LocationManager` component and then registers a new callback that is invoked whenever the GPS location changes. In general, a method for subscribing to an event receives an instance of the class implementing the callback method and further, optional parameters. For GPS location changes, these additional parameters, for instance, control at which granularity the callback shall be invoked, i.e., for every small movement of the device or only for larger relocations. In the example, updates are requested every 5000 milliseconds if the distance has changed by at least 10 meters. For a data flow analysis tool, this additional information can safely be ignored as it does not influence whether a data leak happens or not. It only gives additional information about the circumstances of the leak which is not in the focus of the analysis.

To be completely precise, a static analysis tool that looks for calls to subscription methods would have to maintain a list of all methods in the Android SDK that register callbacks. Such a comprehensive list would not only be large, it would also have to be updated for every new version of the Android operating system. Given that checking whether a certain API method registers a callback or not is a manual effort, maintaining such a list of subscription methods is practically infeasible. Instead, FlowDroid over-approximates this list based on the observation that every callback subscription method must at least accept the callback interface as one of its parameters. We therefore re-formulate the problem as maintaining a list of callback interface types (i.e., fully-qualified names of callback interfaces). Whenever a method takes such a type as a parameter, we assume it to register the respective interface implementation as a callback. In Android, there is only a small number of interfaces. Furthermore, all callback interfaces start with "On" and end with "Listener". This enables us to perform a simple grep over the Android API documentation or source code whenever a new Android version is released to obtain the up-to-date list of callback interfaces. This heuristic is assumed to be safe as the coding rules of the Android Open Source Project enforce naming conventions on all Android core framework developers. In FlowDroid, the list of callback interfaces is saved in the `AndroidCallbacks.txt` file. In practice, we have not yet observed any false positives because of over-approximating subscription methods using parameter types.
The call to a subscription method for an event also further restricts the time at which the respective event can be received (and thus the callback can be invoked). If the app, for example, only subscribes for GPS location changes when a low battery event has been received, this restricts the set of possible event sequences. The first location update must then always be preceded by a low battery event. Directly jumping from the `onResume` method of the hosting activity to the location update is not possible without the low battery callback in between. This problem of possible event sequences has been discussed by Yang et al. [151]. In FlowDroid, we do not capture such dependencies and instead conservatively over-approximate the set of possible events. We assume that every event can happen at every time during the runtime of the respective host component. With similar reasoning, we also chose not to model calls to API methods that unsubscribe an app from events. Instead, we assume that events are never unregistered. In total, event subscriptions are considered to be always active between the `onResume` and `onPause` lifecycle events of the hosting component if they are being subscribed to anywhere in that component. In practice, we have not yet observed any false positive due to this inaccuracy in a real-world app.
One issue, however, arises when calling event callbacks from pre-defined positions within the dummy main method. Anonymous inner classes may reference final variables from their enclosing methods as shown in the example in Listing 41. In this example, the `onCreate` method registers a new button click callback handler as an anonymous inner class which leaks data from a final variable of the enclosing method once the button is clicked. In the bytecode, the anonymous inner class becomes a normal class that receives a reference to the outer class as a constructor parameter. The reference to the final `data` variable is modeled through a second constructor parameter. When creating an instance of the callback class in the dummy main method, the original meaning of this second parameter is lost and there is no longer a connection to the original data it received. Recall from Section 4.15 that parameters of method or constructor calls in the dummy main method are generated artificially. In the example, the inner class will be constructed with an empty string for the `data` parameter instead of the actual source data. Consequently, no leak will be detected by FlowDroid. To properly handle such dependencies and ensure that such leaks are detected, the computation of the respective values (the `data` variable in this case) would have to be copied over to the dummy main method. Only then, the correct values would be available when invoking the constructors of the inner classes. This would require a full backward slice on all variables the callback depends upon. As this would result in significant additional computational effort, we chose to not model such dependencies. In practice, sensitive data is usually not passed in this way, but rather obtained and leaked in the callback as one operation. If sensitive data is passed between callbacks and other code, it usually happens through explicit fields in the outer class.

5.3.2 Dynamic Broadcast Receiver Registration

As described in Section 5.1, an Android app consists of various components that are declared in the `AndroidManifest.xml` file. Broadcast Receivers are special as they can also be registered dynamically, i.e., the list in the manifest file is not necessarily exhaustive. This dynamic extension feature is not available for any other component type. Conceptually, this means that content providers can be divided into two groups: Those known from the manifest that are handled as described in Section 5.1 and those added at runtime. For the latter, FlowDroid must scan for registration methods in the very same way to scans for callback subscription methods. The only difference is how the dummy main method is transformed when a new dynamically-registered instance is detected. For a callback, a method call is added between the `onResume` and `onPause` lifecycle events of the host component to

```
1  void onCreate() {
2    final String data = source();
3    Button button2 = (Button) findViewById(R.id.button2);
4    button2.setOnClickListener(new View.OnClickListener() {
5      @Override
6      public void onClick(View v) {
7        sink(data);
8      }
9    });
10 }
```

Listing 41: Callback in Anonymous Inner Class

simulate the respective callback happening while that component is running. A dynamically-registered broadcast receiver, on the other hand, becomes a new top-level component. Consequently, FlowDroid creates a new section in the dummy main method from the same blueprint that was also used for the statically-declared broadcast receivers.

5.3.3 Method Overwrite Callbacks

In addition to explicitly-registered callbacks and broadcast receivers, one can also implicitly divert the app's control flow from the Android operating system into user code by overwriting framework methods. In Android, users implement components by creating own classes that inherit system-defined base classes. This fact is already exploited when users overwrite well-documented lifecycle methods such as `onCreate`. Additionally, users can, however, also overwrite arbitrary other public or protected methods that have never been intended for the use of app developers. These methods exist because of the internal design of the Android SDK. Consequently, these additional methods are not part of the documented component lifecycle and may change between Android versions without prior notice. Ignoring these methods would, however, yield an incomplete callgraph.
Therefore, FlowDroid scans through all classes that transitively inherit an Android component class (Activity, Service, BroadcastReceiver, ContentProvider) and checks for overwritten methods. The lifecycle methods are filtered out, because they are handled explicitly according to the respective component's lifecycle model as explained in Section 5.1. Calls to these overwritten methods are handled just like dynamically-registered callbacks. At the runtime of their respective component (i.e., between `onResume` and `onPause`), an arbitrary number of calls to these methods is simulated. This disregards the original order in which the Android framework calls these methods and does not consider the circumstances under which they are called. In practice, this approximation is, however, sufficient. Note that this technique is only intended to capture methods that are either not official interfaces (and thus rarely used) or have not yet been explicitly modeled in FlowDroid ("catch-all clause"). Therefore, maximum precision is not the goal here.

5.3.4 Iterative Callback Collection

The above description of dynamically-registered callback handlers and broadcast receivers assumes that the calls to the subscription methods are found by the static analysis tool. For callback handlers, one additionally needs to know in the context of which component the subscription method is called. Otherwise, it would, for instance, not be possible to associate a button click handler with the activity that hosts the respective button. Conceptually, the analysis must thus traverse a callgraph. If a subscription method is transitively reachable from a lifecycle method of a component, then the respective callback belongs to that component. Furthermore, relying on a callgraph ensures that unreachable calls to registration methods are not considered and thus do not produce false positives.
This means that collection subscription methods requires a callgraph. On the other hand, it also extends the very dummy main method from which this callgraph has been constructed. Therefore, the callback handling in FlowDroid has been implemented using an iterative approach. FlowDroid first creates a simple dummy main method that only contains calls to the lifecycle methods of the various components as described in Section 5.1. This preliminary dummy main method is then used to compute a first (unsound) approximation of the app's callgraph. The idea is that the methods in this first callgraph approximation are (transitively) called by the framework. They are the seeds for the control flow inside the app. Only from these methods, the first callback can be subscribed to. FlowDroid then iterates over all reachable methods and checks for calls to callback subscription methods. Whenever a call to a subscription method is found, the target class that implements the callback interface is recorded.

This gradually grows the dummy main method to also emulate calls to the newly discovered callback methods. Conceptually, one needs to include this callback implementation into the callgraph and continue the search for further callback registrations. Note that callback handlers may also register new callbacks, so the search must also continue within newly discovered callback handler methods. This technique gradually expands the callgraph until it covers all methods that can be potentially become reachable at runtime at some point.
For every recorded callback implementation, FlowDroid must also record the component to which it belongs. As explained above, if an activity subscribes for location updates, these updates can only be received as long as the subscribing activity is running. On a more technical level, the association between callback method and component dictates where the call to the callback method needs to be placed in the dummy main method. Ideally, the updates to the dummy main method and to the callgraph would be concurrent: The preliminary dummy main method with only the lifecycle methods is used to create the first iteration of the callgraph. Then, the dummy main method and the callgraph are both extended whenever a new callback subscription is found. In the end, one would have a complete callgraph on which to directly run the taint analysis, and a fully-implemented dummy main method that serves as a model for reference. Unfortunately, the SPARK callgraph algorithm implemented in Soot is not incremental. It was not designed to react to changes in the target program (though purely additive) and extend the callgraph accordingly. Therefore, FlowDroid currently collects all callback subscriptions that are currently reachable, then extends the dummy main method, computes the callgraph anew, and continues the search. This re-computation of the callgraph is a source of inefficiency that could be mitigated by switching to an incremental callgraph algorithm such as the one by Souter and Pollock [125].

5.3.5 Fast Callback Collection

The iterative callback collection is precise, because it can filter out callbacks that are registered in unreachable code. Furthermore, it can precisely associate callback registrations with their respective parent components. On the other hand, this requires iterative extensions to the callgraph which are time-consuming. For cases in which this precision is not needed, FlowDroid offers an alternative callback collection algorithm that favors performance over precision. The idea of the fast collector is to make a single pass over all application classes, i.e., all classes loaded from the analysis target. Only classes loaded from additional libraries on the classpath are not considered. For each class, FlowDroid cans for method override callbacks and then scans over all statements in all its methods to find dynamic broadcast receiver registrations and dynamic callback registrations.
This approach completely evades the need to construct a callgraph. On the other hand, without a callgraph, there is no longer a notion of reachability. Recall that the relation between callback and parent component was based on reachability; a callback is used inside a component if the call site that registers the callback is transitively reachable from a lifecycle method of that component. With the fast approach, this definition is no longer suitable and must be replaced by an over-approximation. We instead associate every callback with every component, i.e., build the Cartesian product of components and callbacks. This over-approximation gives rise to a tradeoff: While the imprecise callback collection as such is substantially faster than the precise one, the dummy main method generated from its outputs is also substantially larger, leading to more possible data flow paths through the program. This can, in turn, increase the runtime of the data flow tracking itself. As we show in our evaluation in Section 8.7, such an over-approximation greatly increases the time and memory consumption of the analysis. In other words, a precise callback collection, though it increases the initial effort, is a substantial requirement for an efficient data flow tracking afterward.

5.3.6 Declarative Callbacks on UI Elements

Callbacks on UI controls such a button click handlers can not only be registered imperatively through subscription methods, but also declaratively in the layout XML file that declares the respective UI element as shown in Listing 42. Recall from Section 5.2 that all layout XML files are transformed into a binary format at compile time. After compilation, the button is only referenced from the program code by its unique numeric identifier instead of the name `myButton`. In Section 5.2, we explained how accesses to such IDs in the program code can be mapped to the respective UI element. For declarative callbacks, the process is more complex: Not the code accesses a UI element, but a UI element registers a callback in the code. There need not even be a reference from the program code to this particular button as such a requirement would violate the declarative nature of callbacks defined in XML. On the other hand, note that the layout file only specifies the name of the callback method, but not the activity class that hosts the button. In general, there is no reference from a layout XML file to its parent activity. Instead, the code of an activity implementation binds to the parent element of the layout (the `RelativeLayout` in the example)

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >
    <Button
        android:id="@+id/myButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="26dp"
        android:text="@string/button3"
        android:onClick="onButtonClick" />
</RelativeLayout>
```

Listing 42: Declarative Callback Definition

```
public class MainActivity extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main); // 0x7f030018
  }
  ...
}
```

Listing 43: Code Reference to View from XML File

at runtime by calling the `setContentView` API method as shown in Listing 43. The example shows the Java code; note that the reference to the constant field `R.layout.activity_main` will be replaced with its value `0x7f030018` at compile time. The referenced element and its sub-elements are called a *View*. In the Android terminology, this API method sets the active view of an activity. The `setContentView` API call also maps the method names for the callbacks in the view to the respective methods in the calling activity class. Also note that multiple activities can bind to the same layout for re-using the layout definitions.
The challenge for a static analysis tool lies in creating a static (potentially over-approximating) view of this dynamic mapping between layout XML file and implementing activity class. Technically, this mapping is two-fold as shown in Figure 15. Firstly, the analysis needs to find the one-to-one mapping between ID and layout XML file. Secondly, it must then map this ID to the correct code class(es) which can be arbitrarily many. We will now describe how both challenges are tackled in FlowDroid.

Mapping Layout Control Files To View IDs

The ID of a layout control file (which is the ID that gets passed to the calls to `setContentView` in the code) references the name of the layout XML file in the app's resource database. This database is a custom hierarchical data structure which is located in a file `resources.arsc` contained in the app's `APK` file. Figure 16 shows the overall layout of the data structure. It has two top-level elements: A string table and a table of packages. The

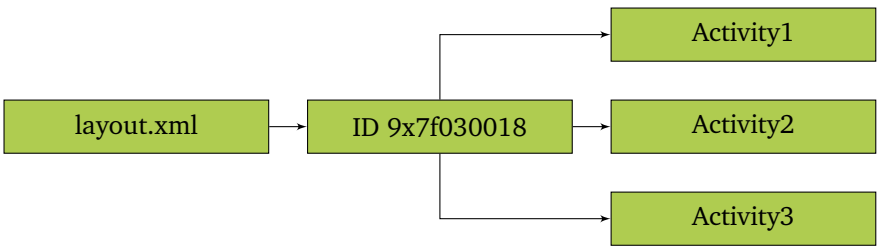


Figure 15: Mapping Between Layout Controls and Activity Implementations

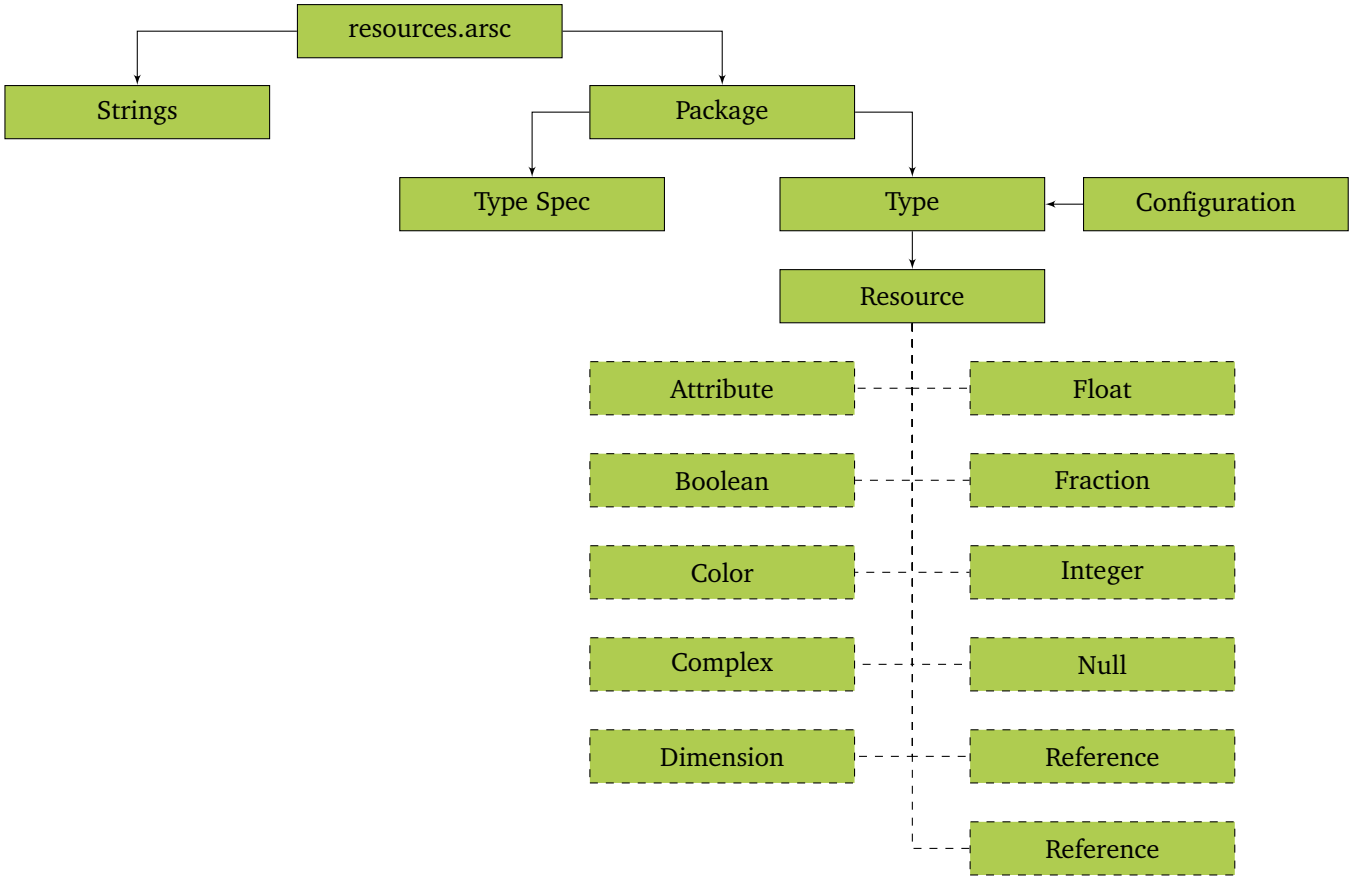


Figure 16: Android Resource File Data Structures

string table is only used internally by the `arsc` database which we will explain later. Each package has a name and a list of declared types. Per package, there is a set of type specifications and of types. For simplicity, we treat type specifications simply as forward declarations for types and do not associate them with a semantic on its own. A type can then be seen as the actual implementation of the type. Each type is associated with one more more configurations. A configuration defines restrictions on the device for which the respective type is applicable. This allows the app developer to, for instance, use different texts for different target locales, or to use different images depending on the screen size and resolution of the target device. Much of the complexity this data structure is hidden from app developers by modern development environments such as Android Studio, though. Types contain an arbitrary number of resources. A resource can be a string, a reference to some other resource, a color, etc. Technically, string resources are references into the global string table.
Each element of the resource file has a unique ID. The ID of a single resource is an integer that represents a combination of the IDs of the containing package, containing type, and the resource-specific ID according to a well-defined bit-shift pattern as shown in Figure 17. For finding the mapping between layout control files and IDs, FlowDroid only needs to read the string resources of the respective type from the resource file. The IDs in the source code exactly correspond to resource IDs in the resources `resources.arsc` file. Therefore, FlowDroid implements the optimization of only querying those layout IDs that appear in the code.

Mapping IDs to Activity Implementations / Source Code

The next step for the overall goal of matching views to host components is to match the layout IDs to the activity implementations that bind to the respective view. Recall that this binding is done through a call to `setContentView` as shown in Listing 43. This means that the task is to map activities to the IDs for which they call `setContentView` at some point during their lifecycle.
Usually, the call to `setContentView` is one of the first statements in the `onCreate` lifecycle method of the activity. This is also how activities are created by common Android development tools such as ADT or Android Studio. In

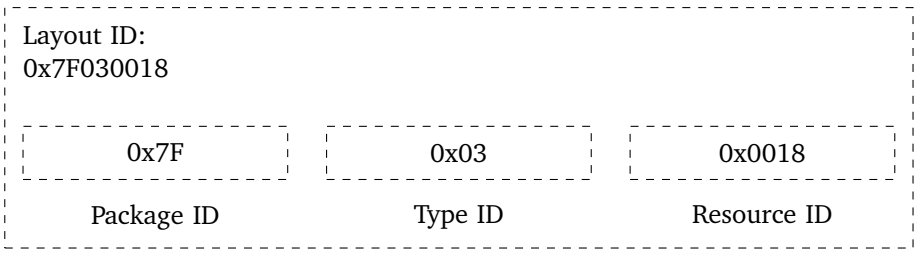


Figure 17: Android Resource ID Layout

theory, an activity is, however, free to switch between views at runtime by calling `setContentView` again later on, or to use different views depending on the outcome of a computation. FlowDroid over-approximates this association. All activities that call `setContentView` with the ID of a particular view at some point are considered to link to that view. If an activity references multiple views, FlowDroid assumes them all to be active at the same time. This design decision over-approximates the set of possible callbacks per activity, but is guaranteed to never miss a callback. Technically, FlowDroid iterates over the bodes of all methods reachable from the lifecycle methods of an activity and looks for calls to `setContentView`. If the argument is a constant integer value, it is directly collected. Otherwise, FlowDroid performs a backward use/def analysis to find the definition of the argument until it reaches a constant value. If the ID of the view is not constant, but calculated at runtime (e.g., due to deliberate obfuscations), FlowDroid cannot perform the mapping. One could conservatively assume that such non-constant view registrations link all possible views to the respective activity, but this can lead to an unacceptable number of false positives.

5.3.7 Callback Sources

Handling callbacks correctly is not only important for obtaining a complete callgraph of an Android app, but also for detecting callback-based sources. In Section 3.2, sources have been defined globally, i.e., certain methods were said to always return sensitive data. In addition to these source methods, data can, however, also be passed from the Android operating system into an application through the parameter of a callback or lifecycle method. In Listing 40, the app is notified whenever the physical location of the device changes. The data object containing the new location (mainly the longitude and latitude) is passed in through the `loc` parameter of the callback and then leaked by the app in Line 7. In FlowDroid, we therefore model all parameters of callback and lifecycle methods as additional sources. For the Android operating system, this conservative over-approximation is also precise as incoming data is usually either sensor data or data received via inter-component communication (i.e., incoming intents).
The only exception in which such data is not relevant and should thus not be tainted is UI callbacks. A button click handler, for instance, receives the button that was clicked through a parameter. The button itself, however, usually does not carry sensitive data as programmers do not commonly store such data in the button's text or tooltip. Since data read from the button is usually not passed to a sink either, the over-approximation explained above does not lead to false positives in the examples we have examined in our evaluation. On the other hand, since apps are highly UI-driven in Android, they contain many UI elements with callbacks and thus many of such parameters that are irrelevant to the taint analysis, but captured by the rule above. Consequently, FlowDroid needs to track the taint status of all those parameters, even though no leak is to be expected, which can have a negative impact on performance. To circumvent such problems, FlowDroid checks whether the type of the callback parameter is a class from one of the Android UI packages, e.g., `android.widget`. If so, the parameter is only considered as a source if the mode for handling UI controls (see Section 5.2) is configured to treat all UI elements as potential sources. By default, only password fields are considered as sources. In this case, callback parameters from other UI widgets can safely be ignored as well.

5.4 Performance Optimizations

The normal FlowDroid data flow tracker is, given appropriatre implementations of the platform-specific interfaces, applicable to all analysis targets that can be converted to Jimple code. Even in its optimizations, it needs to maintain this generality. With platform-specific domain knowledge, on the other hand, one can greatly improve

the performance of the data flow analysis for that particular platform or use case. In this section, we describe such optimizations introduced by the FlowDroid extensions for Android. We also report on optimizations we tried, but that did not prove effective during our experimental evaluation.

5.4.1 Overtaint Filtering

In every static taint analysis, *over-tainting* can occur. In such a case, more data is considered tainted than actually necessary. Besides the increased risk of false positives, this additional taint state can also drastically increase the runtime and memory consumption. Each tainted access path must be propagated over all control flow nodes that (transitively) follow the node that created the taint. In the worst case, a spurious taint is propagated over each statement in the whole app. To reduce the impact of over-tainting, one can generally improve the precision of the data flow tracker, which has limits on its own. Access paths, for instance, are usually (as in FlowDroid) bounded by a specific, fixed length as explained in Section 3.3. All access paths that would exceed this length are truncated and are defined to taint all subsequent field dereferences as well. If the length limit is three, an access path `a.b.c.d.e` is truncated to `a.b.c.d.*`, tainting not only `a.b.c.d.e`, but also all other longer access paths for which it is a prefix such as `a.b.c.d.f`. In this case, the analysis can no longer distinguish the truncated last field access from any other field access with the same prefix and may create false taints if such other fields are accessed.
In practice, the sources of over-tainting in Android apps focus around few root causes. In many apps, activities hold references to other activities. If this activity is tainted completely, it will taint all other activities to which it holds references as well. Since the over-tainting propagates transitively, all objects referenced by those activities will be tainted as well. This can easily lead to significant portions of the app being tainted. In many such cases, the analysis of the app will no longer terminate in realistic time and memory bounds. One reason that can lead to complete activities being tainted are circular references between the activity and a data object. With inner classes, this occurs by default. The activity holds a reference to the data object, which holds a reference to the instance of the outer class that created it (i.e., the activity) in its compiler-generated `this$0` field. Therefore, access paths can grow infinitely large by pointing from the activity to the inner class and back, before reaching the actual field they are meant to point to. When cutting the access path at a fixed length, the result may point to the activity as a whole. While this problem cannot easily be solved without adapting the basic concept of access paths, it can still be easily detected.
As described in Section 4.12.2, FlowDroid supports a memory manager that inspects every newly created taint abstraction before it is propagated inside the IFDS solver. The memory manager has the ability to alter or completely drop the abstraction in case it is inefficient or unnecessary. In the FlowDroid extensions for Android, we provide an extended memory manager that, in addition to the normal access path pooling and compression, checks for hints of over-tainting. If an access path points to a complete component such as an activity or a service, the taint is dropped. While this can lead to a false negative, the analysis would most likely not terminate at all with this over-tainting access path in place.

5.4.2 Callback Filtering

The Android operating system interacts with apps through callbacks as explained in Section 5.3. Through the generated dummy main method, FlowDroid emulates a complete execution of the app including all possible calls to these callback methods. Consequently, the size of the callgraph depends on the precision of the dummy main method. If a callback method is invoked at several spurious places, e.g., a button click is emulated in several components that do not acually host the respective button, these edges must be processed by the data flow analysis. Given that there is tainted data flowing over the spurious edge, one invalid callgraph edge may lead to many more edges in the IFDS solver's exploded supergraph, because the spurious taint is not only propagated into the false callback, but also into all of its transitive callees. Therefore, the performance of the taint analysis can be improved by increasing the precision of the mapping between callbacks and their respective host components. This ensures that the dummy main only calls the callback during the running phase of these components (or, most commonly, the one single component) that actually host it.
Just as in the case of over-taint filtering, increasing the precision of the algorithm that creates the callback-to-host component mappings has its limitations. A callback is assumed to belong to a component if the call site that registers the callback with the Android operating system is reachable from a lifecycle method of the respective component. This check is a reachability problem and is answered by traversing the interprocedural control flow graph which, in turn, is based on the callgraph. Consequently, imprecisions in the callgraph can lead to spurious associations between callbacks and host components. FlowDroid's callback collector provides an interface for *callback filters* to

allow for additional checks. Only if all filters accept a given mapping, it will be used for creating the dummy main method.
FlowDroid implements a number of heuristics in such callbacks filters to remove improbable callback-to-host component mappings. If a callback method is implemented inside a component class, e.g., an activity, this component may only be associated with that particular component. We assume that in apps developed with reasonable software engineering practices and design patterns in mind, developers will not create buttons in activity A that have an `onClick` handler implemented in activity B. This assumption is backed by the fact that such cross-referencing is only possible when registering callbacks by hand in the code. The more convenient declarative registration in the layout XML file that is also generated by the UI editors of popular development tools such as Android Studio requires the callback to be implemented in the current component's class. With similar reasoning, we also restrict callback implementations in inner classes inside component classes to be associated with only the component that is implemented by the respective outer class.

5.4.3 Analzying One Component at a Time

FlowDroid is an intra-component static data flow tracker. Without any extensions such as the ones described in Section 5.6.2, it does not model inter-component communication through intents. The only possible information exchange between two components that FlowDroid does support on its own is through shared memory, i.e., a static field that is written by one component and read by another one. Note that it is not an explicit design decision to model such shared memory communication. It is rather a consequence of how FlowDroid tracks taints through static fields. A static field is always in scope. Consequently, regardless of the component to which the current statement belongs, it can access the static field and the taint can be transferred. Different components are just different parts of the dummy main method, so this taint propagation comes naturally. Consequently, since shared memory communication is not a well-defined feature of the data flow tracker, we can make the simplified assumption that FlowDroid does not handle inter-component communication at all. With this assumption, each component in the app can be analyzed in isolation. Instead of generating one large dummy main method that contains the lifecycle of each component, it is also possible to first generate a dummy main method only containing the first component, fully analyze it, and then repeat the whole process for each subsequent component.
The advantage if this approach is that the dummy main method is significantly smaller, and that the number of taint abstractions that must be kept in memory concurrently is also much lower. The latter is for two resons. Firstly, inside a single component, there is less code that can transfer taints. Since we need to store pairs of statement and taint abstraction for flow-sensitivity, not only the number of tainted variables, but also the number of statements counts. Secondly, inside a single component, there are usually fewer calls to source methods than in the overall app. Consequently, less tainted data is introduces into the app in the first place. Therefore, analzying one component at a time can help analyze apps for which the data flow tracker would otherwise exhaust the available memory before it is able to complete the analysis. We analyze the detailed impact this mode has on analysis time and memory consumption in Section 8.4.

5.4.4 Analzying One Source at a Time

Each source introduces new, unconditionally-tainted data into the analysis that must then be propagated over all statement that are reachable from the source. Taints can be killed early due to strong updates (see Section 4.7), but in general, each configured source negatively affects the performance of the analysis. This effect is multiplied by the number of taints the respective source is called in the code. One trivial approach to improve the scalability of the analysis is, thus, to limit the sources to those the analyst is actually interested in, instead of using one large comprehensive list including all potentially sensitive sources there could be in the Android SDK. While this may increase the risk of missing interesting sources and increases the manual effort for the user to correctly configure the analysis, it also saves time while processing the results, because the analyst need no longer process a potentially large number of uninteresting findings.
In some cases, even a reduced list of sources is, however, still very large. In this case, we exploit the observation, that taints introduced by different sources are independent. Therefore, it is possible to run the analysis with one source at a time and afterwards merge the results. FlowDroid supports such a mode in which it automatically iterates over the sources and runs the IFDS data flow analysis only with one source at a time. Note that this requires special support from the source/sink manager (see Section 4.3). The interface for source/sink managers is extended with iterator-like functions for selecting the next source and checking whether there are any more sources the analysis needs to process.

We observe that the inter-procedural control flow graph, which includes the callgraph and the intra-procedural control flow graphs of all method bodies inside the app, is completely independent of the source currently under consideration. Therefore, we build this data structure only once and make it available to all the different per-source runs. This also means that FlowDroid need not re-construct the dummy main method for each per-source run, which avoids a major performance penalty. Still, we find that this approach does not yield acceptable performance for most apps. Compared to the normal analysis with all sources together, the one-source-at-a-time mode can consume up to 90 times the computation time for apps with many sources. We therefore leave this mode as an idea for future research, but do not encourage its use in practice.

5.5 Related Work

In this section, we present related work that has similar goals as the FlowDroid data flow engine together with its extensions for the Android platform.

Callback and Lifecycle Modeling

In FlowDroid, we explicitly model the Android lifecycle, a technique that other approaches in literature [142] have inherited from the work presented in this thesis. Our callgraph is context-insensitive. When a callback method is called, there is no distinction as to what has triggered the callback. Yang et al. [152], on the other hand, have built an approach for context-sensitive callgraph construction on Android apps. Their approach is based on the assumption that the same event handler can be registered for multiple events (e.g., on different UI widgets) and then acts differently based on the context of the current callback, e.g., which widget was clicked. At the moment, their approach is limited to a handful of event types (creation and termination callbacks). The model that defines these callbacks is still supplied manually. Such manual definitions, even for parts of the model, however. have the drawback that, when a new Android version is released, the model must be updated to reflect the changes. Such changes are usually purely additive and scarce, but can occur. In version 3.0 (“Honeycomb”), for instance, the fragment API was added. It allows app developers to sub-divide activities into smaller, re-usable parts called fragments. To avoid the manual engineering effort of modeling new or extended features, Blackshear et al. have proposed Droidel [21]. Droidel automatically generates application-specific stubs that summarize the behavior of the Android libraries for that app. This stub exposes a single entry point which takes the place of FlowDroid’s dummy main method. Droidel assumes that all of the (current and future) relevant behavior in the Android operating system is implemented in Java code and statically analyzable with sufficient precision. EdgeMiner by Cao et al. [29] is an approach to automatically detect control flow transitions through the Android framework code. Their core idea is to run a static analysis on the entire Android framework once and store the discovered control flow edges as summaries that can be plugged into the individual app analyses.
The authors of DroidSafe [60], on the other hand, argue for manually-crafted stub implementations of the Android operating system. These stubs take the place of the original system libraries during analysis. As the stubs are implemented by human analysis experts, the stub developers can avoid constructs that are hard to analyze statically or that could potentially lower the precision of the analysis. On the downside, this approach requires far more expert labour than Droidel or FlowDroid. The authors state that they had to emulate (i.e., re-implement in Java) 3,176 native methods, 45 classes of proprietary code, and simplify 117 classes in the Java standard library and the Android system libraries. All these classes and methods need to be checked and potentially extended (and/or augmented with new classes and methods) for every new release of the Android operating system.

UI Control Handling

UI controls can be sources of sensitive data as explained in Section 5.2. UIPicker by Nan et al. [101] and Supor by Huang et al [66] are approaches for automatically identifying those UI elements into which users tend to input sensitive data. Their UI models go beyond FlowDroid’s three-fold distinction between (1) ignoring all UI elements, (2) considering only password fields as sensitive, and (3) considering all UI elements as sensitive.

Reflection Handling

Many frameworks and tools for static analysis cannot handle reflective method calls, or can only resolve trivial cases in which the target class and method names are constants. The problem of resolving reflective method calls is orthogonal to the problem of static data flow tracking, but warrants a brief discussion. Li et al. have proposed DroidRA [85, 87] which reduces the problem of finding the runtime values of reflective method calls to a constant propagation problem which is then solved using the COAL [102] solver. The analyzer by Del Vecchio et al.[138] first conducts a backward slice starting at the point where the string of interest is used, i.e., the reflective method

call, and then performs abstract interpretation on the obtained slice. Harvester [112], on the other hand, is a hybrid approach that can extract arbitrary runtime values from Android apps through a combination of static backward slicing and dynamic execution. Harvester first extracts those statements that contribute to the computation of the value of interest. These statements are then used to construct a new app in which exactly this code is run in isolation, ignoring all evasion techniques such as emulator checks that might have been present in the original app. Once the value of interest has been computed, it is written into a log file.

Data Flow Analysis

FlowDroid is based on taint tracking. We tried to implement and evaluate trade-offs that made the analysis applicable to real-world app stores without requiring fundamental changes to existing apps or infrastructure. Other approaches such as the IFC type system by Ernst et al. [43] or the RSCP analyzer [90] are geared towards high-assurance app stores that rather force the app developer to cooperate and than miss potential violations of the data flow policies. For such stores, not accepting an app, because it cannot be proven to comply with the data flow policy, is acceptable, and it is the duty of the app developer to modify his app in such a way that the app store is able to verify it. This can, as in the case of Ernst et al., include the requirement for the app developer to upload source code instead of only the binary app, and to furthermore annotate his source code with a flow policy. The store then only checks the flow policy against the code, and in case of a match, checks whether that policy is acceptable given a store-wide or user-specific reference policy. We, on the other hand, argue that such approaches, while they can give formal security guarantees, inherently restrict the developers. To be fully sound, such approaches must refrain from concepts that cannot be analyzed statically such as dynamic code loading, reflection, and native code. Forcing developers to restructure their apps around such restrictions and, in addition, provide the required annotations is, in our opinion, only possible for restricted domains such as military contract work, but not for a general-purpose app store. General-purpose app stores face competition[27] and overly restricting developers has the potential to scare them away to other competitors. One possible comprise could be to require developers to submit test cases, in the hope that they create these test cases anyway for ensuring the functional quality of their apps. Bastani et al. [19] propose to use these test cases to limit the static analysis to those code parts that are reachable by tests to reduce the code size that needs to be analyzed as well as the number of false positives. Additional instrumentation injected into the app makes sure that the app is terminated when it attempts to execute code that was not covered by any test case and, thus, has not been analyzed. Their strategy is an interactive one. The analyzer is sound, and when it fails to verify a program, the developer has the chance to provide additional test cases that more completely cover the reachable statements in the app.

Huang et al. [68, 96]'s approach is also based on type systems, but with a focus on precision. Their type system DFlow is context-sensitive, but flow-insensitive. For automatically inferring the type specifications, they propose the DroidInfer algorithm based on CFL-reachability [116]. This technique by Tom Reps reduces a number of static analysis problems to graph reachability problems, in which a path between two nodes is only considered to be valid if the concatenation of the labels along this path is a valid word in a context-free language. In fact, Reps describes the IFDS framework on which FlowDroid is built as a particular instance of CFL-reachability. In Huang et al.'s work, each use of a tainted variable amounts to a type constraint. All type constraints together build the context-free language. Only valid sequences of data uses, however, correspond to valid words in that language.

Anadroid by Liang et al. [88] is a static analysis framework for Android that also features static taint flow analysis. Anadroid uses a pushdown system to model dynamically-dispatched interprocedural and exceptional control flow. They use a formal language that closely resembles the Dalvik bytecode language, and for which they then define a formal semantics that manipulates the push-down automaton. They use existing techniques from Reps [118] to compute reachability in the pushdown system, i.e., enumerate the reachable control states. These states can afterwards be checked against a policy to find unwanted data flows. To cope with the large number of events and especially potential interleaving of events in Android, Liang et al. use a technique called entry-point-saturation (EPS). The key idea is to run the analysis on one callback, take the results, use them as the starting state when procssing the next callback, and *widen* the result with the new information from that next callback until a fixed point has been reached [95]. A approach based on model-checking has been proposed by Song and Touili [124]. They define unwanted behaviors in Android apps using Computation Tree Logic (CTL) or Linear Temporal Logic (LTL), to reduce data flow checking to CTL/LTL-based model checking. Van der Merve proposed an extension to Java Path Finders (JPF) to traverse, statically simulate, and verify all execution paths through the app [94].

[27] For Android, there are various third-party stores such as the Amazon store (default app store for all Amazon phones). In countries such as Russia and China, the landscape is even more diverse, with Google Play not even being available in China.

HornDroid by Calzavara et al. [28] abstracts from the semantics of the app using SMT formulas which are then solved by an off-the-shelf SMT solver and is accompanied by a formal soundness proof.
PIDGIN by Johnsol et al. [70] provides a generic query language for program dependence graphs, the same data structure also used by JoDroid [97]. Since PIDGIN allows a wide range of policies to be expressed over the PDGs, it not only supports data flow analysis, but can also check for a variety of common security vulnerabilities in Android Apps at the cost of requiring the user to correctly formulate the corresponding queries. Standard analysis tools that check for privacy violations also usually assume all leaks to be equally important, regardless of the type of the information that was leaked, and regardless of whether the data was leaked in full or only partially. Such a partial leak could, for instance, be a substring of an identifier. While the IMSI number, for example, unique identifies a user, its first three digits only represent the country of the mobile network operator that issued it. Leaking this partial information is thus a much lesser privacy infringement. Barbon et al. [17] therefore propose an analysis that quantifies information leakage. MorphDroid by Ferrara et al. [51] has a similar goal, but also accounts for composite data (e.g., a GPS position being composed of logitude and latitude) and semantic transformations such as associating an address with a GPS position. They use their quantitative model to distinguish accepted leaks such as sending out the user's city name for obtainting a weather forecast from leaking the exact location data which is unnecessary for the purpose of a weather app. In their DAPA work,

5.6 Extensions to FlowDroid for Android

While FlowDroid is a general-purpose data flow engine, it does not directly solve some of the higher-level questions an analyst might have with respect to an app. This section presents approaches that extend FlowDroid to answer such higher-level question. The approaches here build on FlowDroid, rather than replace it or provide different options for the algorithms used in FlowDroid.

5.6.1 Detecting Malicious Flows

Analysts are usually interested in the trustworthiness of an app. FlowDroid can enumerate the data flows inside that app, but it cannot judge whether these data flows constitute expected behavior or malware. Such a judgement usually requires the contextual knowledge of a human analyst as the very same data flow can be completely expected in one app, whereas it is a strong indicator for malware (or spyware) in another app. Assume that an app sends the user's location to a remote web server in constant time intervals. In most applications, this is considered a privacy violation. In a GPS navigation app, this is, however, expected, as the user wants to constantly know his current location on the map to adapt his driving directions.
While manually judging data flows is feasible for individual apps, it does not scale to the market level with thousands of apps. On would have to manually check the data flows of every app that is either freshly updated to the store or updated to a new version. Clearly, automation is necessary for such a scale. To tackle the problem, Mudflow by Avdiienko et. al [14] rephrases the original question. Instead of asking whether a certain flow is malicious, it asks whether a certain flow is uncommon for a certain category of apps. In the example, the question would therefore be "Is it common for a GPS app to send out location data?" in comparison to "Is it common for a notepad app to send out location data?". In the first case, the respective flow will be detected in the majority of apps from that category. In the latter case, such a flow will instead be an outlier with regard to the majority of apps from that category. If one assumes that most apps are benign (and therefore most flows in them are benign), this gives a good indication of possible malicious data leakage in an app.
Another approach to judging whether a data flow is expected by the user of the app or not is AppIntent [154]. While not based on FlowDroid, it also uses static taint tracking to first collect all paths through the app on which data can be leaked. It then attempts to find the root cause for the transmission, i.e., a sequence of user inputs and interactions. This sequence is then given to a human analyst for a final judgement. Though this apporach is not fully automatic and requires a final decision made by a human expert, it greatly reduces the effort of this expert in comparison to a full manual judgement of all flows.

5.6.2 Inter-Component and Inter-App Analysis

FlowDroid is limited to tracking data flows inside a single component within an Android app. Modern apps, however, contain a large number of components. Every screen shown to a user is usually modeled as a distinct activity. Due to this model, the Facebook application, for instance, has more than 300 activities. Additionally, many apps also contain multiple broadcast receivers, services, and content providers. Data is often not only processed

inside a single one of these components, but passed between them. The Android operating system provides a built-in remote procedure call (RPC) mechanism for this purpose called `Binder`. The purpose of the Binder service is to pass messages, called `Intents`, between components. A component that receives such an intent can act to it and further process the data contained in the intent.
For a static data flow analysis tool to track data passing over intents, it must be able to correctly identify the targets of an intent. In other words, one needs to construct an inter-component control flow graph [104]. While this is easy for intents that directly contain the fully-qualified class name of the receiver (called *explicit intents*), it is non-trivial for intents dispatched by the Android operating system (called *implicit intents*). An implicit intent does not contain a pre-defined receiver class. Instead, it only defines metadata about the request such as an *action* string. Components inside the sender app or arbitrary other apps installed on the same device can register to receive intents containing specific actions. If an app sends an implicit intent and there is only one possible receiver registered for the respective action, the intent is automatically forwarded to that component. If multiple receivers are possible, the Android OS asks the user to select the intended receiver. If a user, for instance, clicks on a link inside a document, such an implicit intent for opening the URL from that link is started. If multiple browsers are installed on the device, the user can choose which one shall be taken.
Implicit intents make it generally impossible to always pick the single receiver for an intent that is actually used at runtime. Instead, the analysis needs to conservatively assume that all components that are registered for the respective action are possible recipients of the intents. While such a conservative over-approximation is usually feasible within a single application, one quickly encounters scalability issues when assuming that an intent can be received by components inside other apps as well. In the example with the web browser, one would need to assume that all (possibly unknown) web browsers in the world could receive the respective intent. If one limits “all web browsers in the world” to the scope of an app store, i.e., only considers browsers available inside the store, the list of recipients can still be extensive. Therefore, one usually limits the scope to not look for data flows from a certain app to all possible recipients a user could have installed on his phone, but rather exactly enumerates the concretely installed set of apps. Note that one does not need to consider inter-component communication if one is only concerned about outgoing traffic from a specific app without considering how the data is further used by the recipient. In such a case, one can simply define all methods that send intents as sinks and record the data leaving the app through an intent.
If true inter-component data flow tracking is required, it is important to obtain the intent values used for dispatch (class name for explicit intents and action string for implicit intents) as precisely as possible. The fewer possible values a static analysis needs to consider, the fewer potential recipients need to be analyzed. Not knowing the action string of an implicit intent at all due to analysis imprecision can be considered the worst case; every component in every app on the device could be the potential recipient. IC3 [103] tackles exactly this problem. It uses *composite constant propagation* to find not only precise values, but also precise complex structures consisting of multiple values. Support for complex objects is important for inter-component communication on Android as one can not only use the action string described above, but also various other values (and combinations thereof) to filter potential receivers.
For combining the intra-component data flow analysis provided by FlowDroid with the inter-component control flow graph created by EPICC [104], two different techniques can be used. ICCTA [86] first creates the inter-component control flow graph and then combines all the different components of all apps under consideration into one big single component. All IPC calls (i.e., calls that send intents) are replaced with direct method calls. Instead of sending intents through Android’s Binder mechanism, the intent object is passed to the receiver method of the target component as a normal parameter of a Java method call. This simplified app does not need any IPC handling anymore and can directly be analyzed by an unmodified version of FlowDroid.
DidFail [77] uses the reverse approach. It first conducts data flow analyses on each individual components and later combines these individual specifications to obtain the full picture. In the first step, all methods that can potentially receive intents are considered as sources and all methods that can potentially send out intents are considered as sinks. For every component, this gives a full specification of the respective component’s behavior with regard to inter-component communication. Note that the ICC sinks extend the normal set of sources and sinks instead of replacing it. This is important for also detecting cases in which a component is no a middle-man, but an endpoint of an inter-component data flow. In other words, the app either loads data from a source and sends it out as part of an intent, or receives potentially data as part of an intent and passes it into a sink. In total, DidFail creates a set of specifications for each component that are of the following form:

- Source `src` flows to intent `X`

- Intent `X` flows to sink `snk`
- Intent `X` flows to intent `Y`
- Source `src` flows to sink `snk`

In the second step, the inter-component control flow graph created by EPICC is used to combine the individual component specifications to obtain the full data flows. Conceptually, ICCTA and DidFail compute the same results. Researching the exact benefits of one approach over the other is an interesting subject of future work.
The approaches mentioned so far take a purely static approach to inter-component and inter-app data flow tracking. DroidForce [110], on the other hand, combines FlowDroid's static intra-component data flow tracking with a runtime mapping between components. The key idea to pre-compute all data flows within a component in a conceptually similar way as DidFail. Instead of statically merging these summaries over components, this step is, however, delayed to the runtime of the app. The static data flow summaries are injected into the apps as lookup tables. Additionally, DroidForce augments the app with additional code for reading out and gluing together the right flow summaries at runtime.
Consider a simple scenario with two components, regardless of whether they are in the same app or not. Component A reads data from a source and then passes this data on to component B via an intent. Component B writes the received data into a sink. The instrumented code in component A is triggered just before the intent is about to be sent. It uses the pre-computed static data flow tables to look up which data flows have the current intent-sending call site as a sink. The respective sources of these data flows are considered potential sources of the data being transmitted. This set of potential sources is then appended to the intent before it is sent. When component B receives the intent, the instrumented code of component B is triggered. This code looks up all data flows that begin at the current intent receiver callback in the table of statically pre-computed data flows for component B. It then assigns the received sources from component A as sources to these flows. The resulting flows look as if they had been computed for a combination of component A and B similar to the output of DidFail. More concretely, if component A knows that the phone's unique device ID is sent via an intent, this source information gets appended to the intent. Component B knows from its own lookup table that the incoming intent data is leaked via SMS. When it learns from the additionally- injected intent data that this data was originally the phone's unique device ID, it can combine the two partial paths. In sum, it can conclude that the unique device ID is leaked via SMS: `getDeviceID() -> Intent -> sendTextMessage()`.

6 Automatic Library Summary Generation: StubDroid [29]

As explained in Section 4.9, Android apps as well as Java programs usually rely on large libraries to provide their functionality. In version 4.2, the Android SDK offers over 110,000 public methods and the number constantly grows with every new release. For an analysis to be precise, it must not only analyze the application code, but also consider the effect of these library methods on data flows inside the application. Existing approaches to static analysis deal with library methods in one of three ways. The first class of analyses precisely models a subset of these framework methods manually [49, 54, 60, 75, 91, 155]. This also includes FlowDroid's EasyTaintWrapper component explained in Section 4.9.1. The second class of approaches analyzes the entire Android framework together with every application [90]. The third class uses simple rules of thumb such as "taint return value of call if one or more arguments are tainted", which are expected to cover the most common cases [67]. All of these approaches exhibit serious drawbacks.

Providing hand-written models for the framework is cumbersome to implement and requires manual re-evaluation of (and possibly changes to) the model for every new framework version, which is a prohibitive effort given the size of the code to be understood and modeled. CHEX [91], for instance, opts to not analyze the framework together with the application, and instead resorts to externally-defined models. However, the source of the models is mostly left open, effectively burdening the user with this non-trivial task. FlowDroid's EasyTaintWrapper, for instance, provides models for the most commonly used classes such as Java collection classes or String APIs, but may miss some less-used APIs. Therefore, the analysis may miss leaks or require the user to double-check whether all APIs used in his target app are modeled in the EasyTaintWrapper's configuration file prior to the actual taint analysis.

Including the libraries (e.g. all of the Android framework's code) in the automated code analysis evades this manual effort, but it introduces an often prohibitive overhead on the code analysis in terms of the overall code size to be analyzed. The Android operating system consist of millions of lines of code, thereby greatly exceeding the size of usual applications to be analyzed, causing the analysis to spend significantly more time in analyzing library code than in analyzing application code, i.e., the code that is of actual interest to most analyses. We found such an approch to induce a significantly increased analysis time—for every single app, over and over again. Furthermore, in the case of system-specific libraries, the code might not even be available on the analysts machine, as it is specific to a very specific execution environment, i.e., one smartphone model. This problem is underlined by the fact that Google only ships stub classes together with its Android SDK which are sufficient for type information and to link apps against them, but contain no actual implementation. These stub methods only throw exceptions stating that the respective method is not implemented.

Applying rules-of-thumb instead of a proper library implementation or external model evades the problems of high runtime overhead and high manual effort. However, it remains unclear to what extent these rules actually cover all important libraries, or whether false-negatives may occur due to missing taint propagations. Worse, if these rules are designed to avoid false-negatives, they necessarily need to over-approximate the behavior of method calls and can thus lead to an increased number of false-positives. Recall that these rules are oblivious to the individual callees, but aim at a generic model for all library calls.

In conclusion, none of the approaches presented so far solves the challenge of handling libraries during static analysis to full satisfaction. In this section, we thus present StubDroid, the first fully automated approach for inferring taint-flow models from binary distributions of libraries such as the Android framework or the Java JDK. StubDroid first performs a taint analysis on selected public methods. We demonstrate this for the collections API, which is the most commonly used part of the framework. StubDroid then generalizes the resulting source-to-sink data-flow mappings and stores them in a summary file. When a static client-side taint analysis later processes the invocation of a framework method within the code of an Android app, the analysis can simply plug in the information from the summary file, and short-circuit further analysis of the framework call and all its transitive callees. While seemingly simple, a significant challenge lies in establishing summaries that are field sensitive and handle aliasing, and in treating correctly the various callbacks that the framework code might use to interact with the application. Furthermore, note that the generated summaries are independent of any concrete client application. They must thus abstract from all possible state or call sequences while still retaining maximum precision.

We show that on commodity hardware it is usually possible to generate library summaries in under three minutes per framework class— a one-time effort. Using StubDroid's summaries for conducting a taint analysis on client applications with the FlowDroid taint tracking tool can improve the analysis performance by over 90%. For many applications, the use of summaries even *enables* the analysis, as the full analysis of those apps would time out after

[29] Large parts of this Section are taken (directly or with minor modifications) from our 2016 ICSE paper[5]

spending 30 minutes and consuming tens of gigabytes of memory if summaries were not used. StubDroid avoids these blowups because, opposed to client analyses, its analysis can focus on one framework entry-point method at a time.
In summary, our work on StubDroid provides the following research contributions:

- StubDroid, a method for automatically generating correct and precise models of the Android framework methods with respect to taint analysis,
- a full open-source implementation of the above,
- as an artifact, taint summaries for various important Android APIs, and
- an evaluation of the peformance effect of summary usage in the FlowDroid taint-analysis tool.

Note that the StubDroid approach is conceptually generic and can be applied to all taint analyses based on access paths. Our current implementation is based on FlowDroid and optimized inside the Soot and FlowDroid frameworks. StubDroid's full source code and the library summaries generated with it are publicly available as an open-source project at: `http://blogs.upb.de/sse/tools/stubdroid/`
The remainder of this section is structured as follows. In Section 6.1 we motivate why simple rules of thumb are not sufficient to handle library methods in a taint analysis. Section 6.2 shows what taint summaries for a method look like and also introduces the concept of access paths which StubDroid uses to model field references. Section 6.3 explains StubDroid's architecture for generating and applying summaries, before we report on our summary-computation process in detail in Section 6.2. Section 6.5 focuses on StubDroid's callbacks handling, before we go into the process of applying summaries in Section 6.6. In Section 6.7, we report on the performance and correctness of StubDroid. Section 6.8 presents related work.

6.1 Motivating Example

Using summaries, static analyses can gain considerable performance improvements. A summary, plugged in at a call site, not only renders unnecessary the analysis of any direct callees, but also of all of the methods called transitively. A summary thus truncates a whole call tree and replaces it by a single leaf containing the summarized data-flow facts. Even the summary for a simple method such as `HashSet.add()` will shortcut the analysis of about a dozen methods as shown in Figure 18. When analyzing the client program, the whole tree is flattened into a single rule: "For a HashSet `s` and an element `x`, if `x` is tainted, then `s` is tainted after executing `s.add(x)`." Given that `HashSet.add()` is used many times in many applications, this can save considerable analysis time.

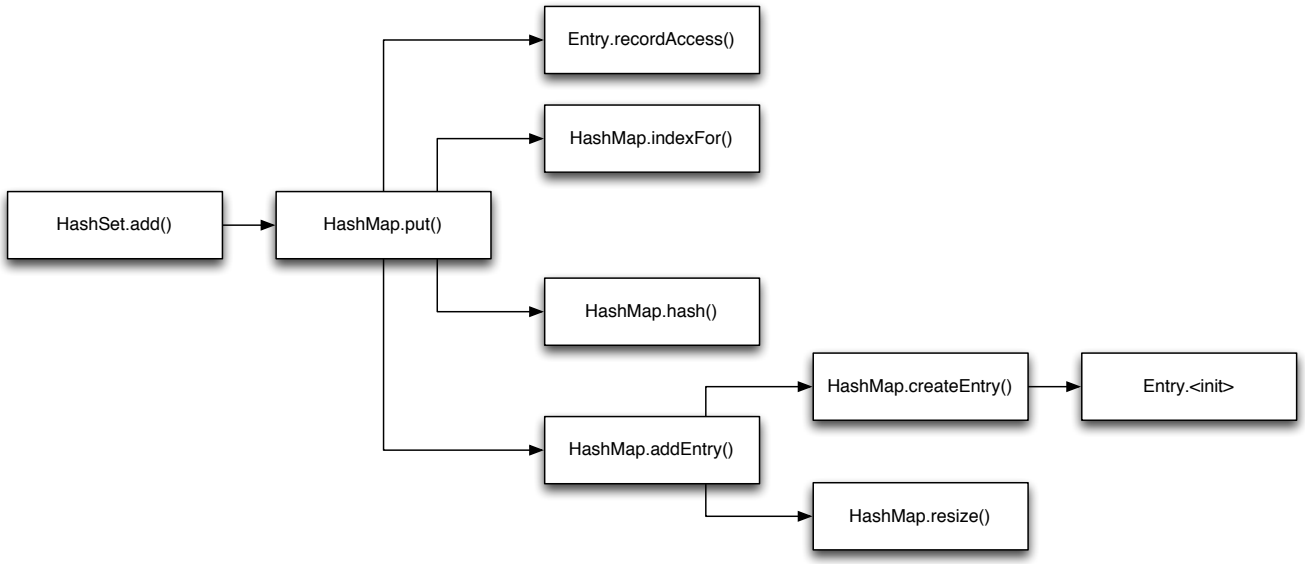


Figure 18: Call Tree for `HashSet.add()`

```
1 public void doLeak() {
2   ByteArrayOutputStream out = new ByteArrayOutputStream();
3   ObjectOutputStream oos = new ObjectOutputStream(out);
4   oos.writeObject(source());
5   oos.close();
6   sink(out.toByteArray());
7 }
```

Listing 45: Complex Inter-Object Tainting

It is important, however, that summary rules are sufficiently precise, as imprecisions can carry over into the analysis result and can again degrade analysis performance. Ad-hoc rule sets often fail to distinguish different fields of the same object or different parameters of the same method call. A simple generic rule as above generally taints, whenever invoking any method `o.m(x)` with a tainted parameter `x`, the base variable `o`. As we show in Listing 44, such a coarse model can easily yield imprecisions, and therefore false positives. In this example, upon processing the constructor call at line 19, the analysis would taint the variable `p`, and implicitly all fields reachable through it. In the example, this would cause a false positive at line 20: as the return value of `get01()` is retrieved from the tainted reference `p`, this return value is (falsely) considered tainted as well.

```
1  public class Pair {
2    private Object o1, o2;
3
4    public Pair(Object p1, Object p2) {
5      this.o1 = p1;
6      this.o2 = p2;
7    }
8
9    public Object get01() {
10     return this.o1;
11   }
12
13   public void setComplex(Data a) {
14     this.o1 = a.b.c;
15   }
16
17   public static void main(String[] args)
       {
18     String s = source();
19     Pair p = new Pair("not tainted", s);
20     sink(p.get01());
21   }
22 }
23
24 public class Data {  public Data2 b;  }
25 public class Data2 { public Object c; }
```

Listing 44: Complex Data Structure and Method

In some cases, too simplistic rules can also lead to false negatives such as in the example in Listing 45. In this example, `oos.writeObject(..)` (line 4) writes data to `out`, via an internal field reference. The standard rule set would mark `oos` as tainted as well as all fields reachable through it, but not `out`. But the data is leaked through `out`, causing analyses using such a summary to miss the flow to the `sink`. A sound summary must thus be able to encode that the call to `oos.writeObject(..)` has an implicit side-effect on (internal fields of) `out`.

As these examples show, too simplistic rules can lead to serious cases of over and under-tainting. Nevertheless, simplistic rules are the current state of the art [9, 60, 91, 155]. The only simple alternative to summaries, however, would be to analyze the complete Android SDK together with every app being inspected. Past experience as well as our experiments in Section **??** show, however, that the performance penalty of such an approach is prohibitive. StubDroid thus seeks to pre-compute precise summaries ahead of time, in an automated one-time effort. These summaries can then speed up any subsequent analysis runs without jeopardizing precision, and while reducing memory consumption.

Library summaries model the effects of library methods on data accessible to the client while abstracting from all the internal processing of the library. This allows client programs to be analyzed even if the library implementation is not present. Even more importantly, by producing summaries, the library and all its transitive dependencies only need to be analyzed once. After a summary has been computed, it can be applied for an arbitrary number of client program analyses; which is especially useful if the library is common and large, such as the Java collections API.

6.2 Summary Model

Assume the `Pair` class from Listing 44 to be part of a library. The summary for the `Pair` class' constructor needs to model that data flows from the first parameter to the field `this.o1`, i.e., that `this.o1` inherits the taint state of the first parameter. The same connection exists between the second parameter and the field `this.o2`. There is, however, no connection between the two fields or between the first parameter and field `this.o2`.
The taint summaries generated by StubDroid take the form of rules. Given a certain incoming taint, they model the effect of a certain method call on this taint. The constructor of the `Pair` class can therefore be described using two rules:

1. `this.o1` is tainted if `parameter 1` is tainted. (R1)
2. `this.o2` is tainted if `parameter 2` is tainted. (R2)

When applying a summary, the rules are used to perform a fixed-point iteration on the set of tainted variables at the call site that invokes the summarized method. Every taint state that is not explicitly changed by a summary rule is kept unchanged. (There are no strong updates.)
In Listing 44, which calls the constructor of `Pair` with only `Parameter 2` tainted, summary rule (R2) is applicable, while rule (R1) is not. Thus, `o2` is marked as tainted for the current instance of `Pair` while `o1` remains untainted.
Note that summaries can also model effects on private fields of objects. While those fields are invisible to application code, they can be used to model object state, akin to ghost fields in JML [79]. This makes summaries field sensitive, which is important for maintaining precision. The `Pair` class in the example contains a method `get01()` returning `this.o1`. The return value of this method thus inherits the taint state of `this.o1`, which is distinct from the taint state of `this.o2`. Without field sensitivity, one could not distinguish between the taint states for the two fields, and a false positive would occur at line 20.
Highly precise data-flow analysis tools such as FlowDroid or Andromeda [135] work by tracking not only fields but so-called *access paths* as explained in Section 3.3. Recall that an access path is of the form $l.f.g$ where l is a local variable or parameter and f and g are field accesses. Access paths can have different lengths up to a user-customizable maximum, at which they are truncated. An access-path of length 0 is a simple local variable or parameter, e.g., l. Truncated access paths act as placeholders for all runtime objects reachable through them, and end with an asterisk, e.g., $l.f.*$ for all objects reachable through $l.f$ (including $l.f$ itself, but also $l.f.g$, $l.f.h$, ...). To retain this level of precision, StubDroid's summaries are also based on access paths, with a customizable maximal length. A taint-summary rule thus always taints an acess path given that a certain incoming access path is tainted. One would thus write the above rules more precisely as:

1. `this.o1.*` is tainted if `parameter 1.*` is tainted.
2. `this.o2.*` is tainted if `parameter 2.*` is tainted.

In all rules, the asterisk character acts as a placeholder. Assume an application calls the constructor of `Pair` as in:

```
Data d = new Data(); d.f.g = source();
Pair p = new Pair("not tainted", d);
```

In this example, rule 2 applies. It will, however, not simply taint `this.o2.*`, but copy over the suffix of the access path, tainting instead `this.o2.f.g`. More formally, the asterisk on both sides of a rule references the same universally quantified variable ($\forall x$: `this.o1.`x is tainted if `parameter 1.`x is tainted). In result, StubDroid's summaries can retain the client analysis' precision as long as the library summary was generated with at least the same maximal access-path length that is also used to analyze the application. By default, access paths are truncated at length 5, the same default that also the FlowDroid client analysis uses. Longer access paths can require significantly more computation time during summary generation, but are possible by simply changing the StubDroid configuration. In previous work, we found FlowDroid's default length of 5 to be sufficiently precise in practice [9].

6.3 Architecture

Figure 19 shows the general workflow involving StubDroid. StubDroid generates one summary file per library class, from a binary distribution of the respective library. At this stage, no application code is present. We assume

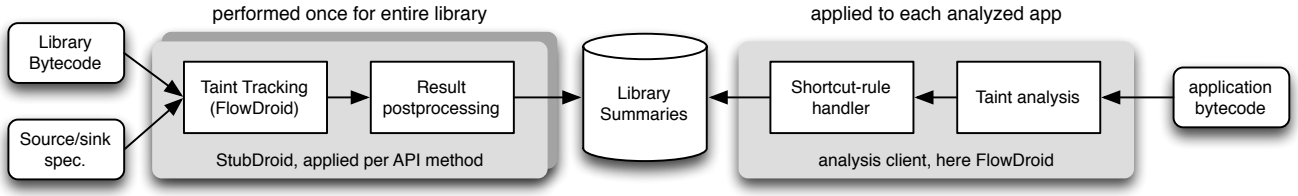


Figure 19: StubDroid's Process and Architecture

that applications only interact with libraries through public methods and fields, rather than applying reflection to access private members. Therefore, all public methods (and only those) need to be summarized.
The summary generator cannot anticipate in which order or with which parameters the library method will later be called inside the application code. The generated summaries must work in all possible apps and usage contexts. Therefore, StubDroid must assume all possible call sequences, must abstract from all library state and parameters before a method call, and must then analyze the effect of the method on these abstract descriptions as explained in Section 6.2. StubDroid can thus analyze the effects of every API method in isolation, which helps the tool to keep its memory requirements low despite the analysis' high precision.
StubDroid generates summaries as XML files, one file per library class. The generated summary files can afterwards be used during the analysis of an arbitrary number of applications. At this stage, the library code no longer needs to be available, as the summaries are self-contained. When using the libraries during an analysis, the XML files are loaded on demand. A client-side summary storage keeps track of all classes for which summaries are available. Only when a taint can potentially reach a field or method of a certain class, the summary file for this class is requested from the storage and loaded into memory. This greatly reduces memory consumption if summaries for large (or many different) libraries are available on disk.
In our particular implementation, StubDroid uses the FlowDroid open-source data-flow tracker for generating and as a client applying library summaries. Nevertheless, the summaries are encoded in generic XML, which makes them usable also for other taint-analysis clients. We based StubDroid on FlowDroid, because it is precise (context-, field-, object-, and flow-sensitive) as well as easily extendable. Further, as FlowDroid is based on Soot [78], it can be run on Java source code, Java bytecode, and Android's Dalvik bytecode[30]. StubDroid inherits these capabilities.
To integrate library summaries into its taint analysis, FlowDroid offers the concept of so-called *Taint Wrappers* (see Section 4.9). Those wrappers implement shortcut rules that circumvent the analysis of certain callee methods. To enable FlowDroid to process StubDroid's summaries, we implemented shortcut-rule handling as a taint wrapper which obtains the summary rules directly from the summary storage.

6.4 Summary Generation

StubDroid generates data-flow summaries and must thus conduct a data-flow analysis on the library's bytecode or source code. As explained before, StubDroid analyzes the library by focusing on one API method at a time. Given such a method, StubDroid conducts a taint analysis starting at a number of different potential data sources. In general, every access path within the method in question must be considered a source. This includes all access paths involving parameters, the `this` reference for instance methods, but also all visible static fields. Every access to one of these access paths becomes the left-hand side of a summary rule. For example, in Listing 44, the constructor of the `Pair` class induces rules for the access paths `this.*`, `p1.*` and `p2.*`.
Further, note that `this.*` is used as a source instead of the combination of `this.o1.*` and `this.o2.*`. StubDroid starts with such more abstract top-level access paths only, and reconstructs on demand which fields have actually been used. We will detail this later. Using longer access paths to start with would require StubDroid to consider all access paths reachable through the above base variables, which could cause a combinatorial blowup.
StubDroid, further considers every method return as a sink at which the summarized flow is concluded.[31] The summary rules are generated by comparing the taint state of access paths before and after the execution of the method to be summarized. Every taint derived from one of the source access paths that reaches the end of a method must be translated into a summary rule. In the example from Listing 44, a flow derived from `p1.*` reaches

[30] The new ART runtime uses the same bytecode as Dalvik.
[31] Exceptional method returns are ignored in the current implementation, but do not pose any specific further challenges aside from the implementation effort.

```
<method id="test.Pair: void setComplex(test.Data)">
  <flows>
    <flow isAlias="true">
      <from sourceSinkType="Parameter" ParameterIndex="0"
        BaseType="test.Data"
        AccessPath="[test.Data: test.Data2 b, test.Data2: java.lang.Object c]"
        AccessPathTypes="[test.Data2, java.lang.Object]" />
      <to sourceSinkType="Field"
        BaseType="test.Pair"
        AccessPath="[test.Pair: java.lang.Object o1]"
        AccessPathTypes="[java.lang.Object]" taintSubFields="true" />
    </flow>
  </flows>
</method>
```

Listing 46: Taint Summary for `Pair.setComplex()`

the end of the constructor method as a taint `this.o1.*`. This leads to the generation of rule (R1) from Section 6.2. Note that identity flows are not encoded, e.g., no rule is generated for the flow that starts with `o1.*` and ends with the same `o1.*` at the end of the method. This is because generally StubDroid does not support strong updates to kill flows. All taints that existed at the beginning of a method are implicitly assumed to still exist also after the summarized method's invocation. We will consider strong updates in future work.
As the sources are only top-level access paths, rule generation is not trivial. Assume that at the end of method `setComplex` from Listing 44, an access path `this.o1.*` is tainted. As taint is linked to the source `a.*`, not `a.b.c.*`, this can lead to all of the following rules:

- `this.o1.*` is tainted if `a.*` is tainted.
- `this.o1.*` is tainted if `a.b.*` is tainted.
- `this.o1.*` is tainted if `a.b.c.*` is tainted.

To restrict the rules it needs to generate, StubDroid must find the concrete fields that were accessed during taint tracking. This is achieved by analyzing the concrete taint-propagation path backwards. The taint analysis is configured to record every statement that influenced a taint, i.e., for which the tainted access path changed during the propagation. For method `setComplex`, this generates the following taint-propagation path. (The temporary variables are artifacts of the Jimple-representation [78] that StubDroid operates on.)

```
tmp$0 = this.a;
tmp$1 = tmp$0.b;
tmp$2 = tmp$1.c;
this.o1 = tmp$2;
```

As the path shows, the taint `this.o1.*` was derived from `tmp$2.*` which in turn was derived from `tmp$1.c.*` etc. Note that this backwards-analysis makes taints more precise. Whenever an assignment defines the base variable of the current taint, its right-hand side replaces the current base variable of the access path. On the second statement, the taint `tmp$1.c.*` is mapped to `tmp$0.b.c.*`, because the second statement defines `tmp$1` as `tmp$0.b`. In other words, StubDroid uses assignments to extend the access path with the field information from the right-hand side of the assignment. With the first statement, the final result of `this.a.b.c.*` is reconstructed. This is the most precise source access path for the new rule. Therefore the actual rule that StubDroid computes is: `this.o1.*` is tainted if `a.b.c.*` is tainted.

6.4.1 XML File Storage

StubDroid stores its summary facts in one XML file per class. This allows clients to load summaries on demand. Only when a taint reaches a method declared in a specific class, that class' summary file must be loaded. Listing 46 shows the summary of the `setComplex()` method from the example class in Listing 44. A summary file contains `method` elements which in turn contain `flow` elements, one per pair of source and sink between which the method causes

```
1 int charToInt(char c) {
2   int[] vals = new int[]  { 0, 1, 2, 3, 4, ... };
3   int idx = (int) c;
4   return vals[idx - 48];
5 }
```

Listing 47: Implicit Flow Sample Code

a data flow. For each flow, StubDroid stores the source (`from`) and the sink (`to`) in terms of the method's interface. Sources and sinks differentiate between method parameters, fields, and return values. Method parameters are referenced by index. Fields are represented by their full signature. If not the parameter or field itself, but an access path starting at the respective element is referenced, the fields on the access path are stored as an ordered list of full field signatures.
Additionally, StubDroid further stores the propagated types of all fields referenced in a flow summary. While the normal field signature only contains the declared type of the field, the taint analysis also provides StubDroid with an (often more precise) type propagated along with the respective taint as explained in Section 4.11. When applying the summary, this allows StubDroid to inject these precise runtime types back into the client's taint analysis. This is useful as some clients use type information to refine call-graph information on the fly. A library method, for instance, might be declared to return `java.lang.Collection`, but always return a more precise type `HashSet`. In this case, the taint analysis can make use of this information: when seeing a call `c.add(..)` on a tainted collection `c`, if the client knows that `c` is a `HashSet` then it can resolve the `add`-call precisely to `HashSet.add(..)`.

6.4.2 Summaries and Implicit / Native Flows

The computation of library summaries is based on a taint analysis which can either only be performed for explicit data flows through assignments, or also for implicit flows through control-flow dependencies. Listing 47 shows an implicit data flow within a method for converting characters containing digits to integers. The conversion functions in the Oracle JDK and the Android platform are implemented in a similar fashion. Method `charToInt` does not directly assign the parameter value to the result value, but still the result depends on the parameter, and a summary should contain a rule which taints the result value if the parameter is tainted.
StubDroid offers different possibilities to handle such cases. The FlowDroid data-flow tracking tool that StubDroid uses internally supports computing implicit flows just like "normal" explicit flows if the respective option is enabled as explained in Section 4.6. In many cases, however, implicit flows do not fully leak the data in question but rather leak information about that data. Determining whether this information leak is problematic is a hard semantic problem [76]. Given that tracking of implicit flows is also expensive, we thus recommend not to track them and instead specify manual summaries for the small number of relevant data-type conversion methods by hand. These are basic JDK methods such as `Integer.toDecimalString()` that are hardly ever extended or changed. This is how we use StubDroid in our daily work and how we conduct our experiments.
Similar problems occur when libraries reference native code. In the Oracle JDK's `ConcurrentHashMap` implementation, for instance, native code is used for fast non-blocking access to the underlying data structures. Since the FlowDroid, and thus StubDroid, cannot analyze native code, we created summaries for such methods manually. This is semantically equivalent to using FlowDroid's native call handler (see Section 4.10) that hard-codes summaries for the most prevalent cases. Note that StubDroid does not support sanitizers or flows across external resources such as files. We leave this to future work.

6.5 Callbacks

In general, Java (and Android) methods can contain callbacks in which a library method invokes client code. We distinguish generic callbacks such as `toString()` from the special case of Android lifecycle callbacks.

6.5.1 Generic Callbacks

Every library method can invoke methods on objects passed in from application code to call back into the client code, see Listing 48. A correct summary of method `append(..)` requires knowledge about the concrete implementation of `IAppender` (more concretely: of `IAppender.getString()`) passed to `append(..)`. Without this knowledge, one has to manually choose an approximation. An under-approximation of the effects of the call to `getString()`

```
1 public String append(String inStr, IAppender appender) {
2   return "Hi" + appender.getString(inStr);
3 }
```

Listing 48: Callback Example

would assume its return value never to be tainted. An over-approximation, though, would assume the return value always to be tainted. A probably more useful approximation would be to assume the return value of `getString()` to only be tainted if the `inStr` parameter is tainted when `append(..)` is called. Even such an approximation might be incorrect, though, depending on the concrete possible implementations of `IAppender`.
To handle such callbacks precisely, StubDroid adopts the principle of component-level analysis introduced by Rountev et. al. [119]. During the analysis of a method to be summarized, one may reach call sites for which the library itself contains no callees. We call these call sites *gaps*. The summary rules presented so far connect interface components of the respective library method, e.g., link a parameter to a return value. With callbacks, flows may start and end at gaps as well. A gap can be thought as a "hole" inside a summarized method flow. StubDroid cannot make any assumptions as to what transformations are made to a taint abstraction inside a gap. It can only summarize the part of the flow that is inside the library method until it reaches the gap. For all possible outgoing taints, it can then again summarize how they flow further inside the library method. In other words, a call to a gap method is treated like an additional interface of the method to be summarized. In the example of Listing 48, this leads to the following rules:

- `<Gap1>Parameter0.*` is tainted if `inStr.*` is tainted
- `Return.*` is tainted if `<Gap1>Return.*` is tainted

Note that StubDroid must generate rules for every possible outgoing taint of the gap method. In the example, only the return value of the gap method `getString` is used within `append`. If, however, a gap method is, for instance, called with a heap object as a parameter and that object is used later on, this must also be represented by a flow. This flow would then account for gap implementors that taint fields inside the heap object they received as a parameter. In short, all possible ways in which a taint can be passed back from a callee to its caller can lead to new flow rules.
When the summary is later applied to a target program, the gaps must be filled with either a different summary or with the results of analyzing client code. StubDroid first attempts to find other summaries that can fill the gaps in question without any interaction with the client analysis. If unsuccessful, it passes the taint information at the gap's call site to the client analysis to find additional implementations of the gap method in client code. This allows the client analysis to focus on the client code alone when filling gaps. The fill-ins applied to gaps can in turn have new gaps which must be filled using the same principle. The fixed-point iteration stops if there are no open (i.e., unfilled) gaps remaining.

6.5.2 Android Lifecycle Methods

Lifecycle methods allow the Android platform to control applications. In general, the Android OS is much more tightly coupled with its applications than a normal Java VM with its programs. Android applications do not contain a `main` method, but instead derive classes from certain pre-defined system classes and overwrite so-called *lifecycle methods* which allow the the Android middleware to, for instance, start, pause, or resume applications when necessary.
Currently Android comprises four different types of *components*: Activities, Services, Broadcast Receivers, and Content Providers. All of them have distinct lifecycles, defining ways for the operating system to influence the execution of the respective component. Nevertheless, the semantics of all four lifecycles is rather self-contained and well documented. The framework essentially just calls the implemented fraction of the lifecycle methods in a predefined, well-known order. The most complex lifecycle is the one of the activity component type, and even this one contains fewer than a dozen methods. We therefore leave the simulation of lifecycle-induced call-backs to the summary client. In our experiments, we use the concrete client FlowDroid, which handles lifecycle methods by simulating their effects through a generated dummy main method that simulates the lifecycle of the Android application as explained in Section 5.1.
Notifications are special callbacks that allow the Android operating system to notify applications of system events like a battery shortage or an incoming text message. Sensors like GPS are also modeled through callbacks in the

```
1 void doLeak() {
2   Data data = new Data();
3   data.b.c.d = source();
4   Pair p = new Pair("foo", "bar");
5   p.setComplex(data);
6   sink(p.get01().d);
7 }
```

Listing 49: Client Program for Pair Class

application: When the user moves around, a special interface in the application is called with the new coordinates of devices. These callbacks can be invoked by the operating system at any time while the respective host component is running, and they are provided through about 200 special-purpose interfaces. With our implementation we provide a list of these callback interfaces. To obtain a complete and precise list of callback methods, it is therefore sufficient to match all methods contained in interfaces in this list against the list of interface methods implemented in the target program. Clients must then simulate a call to all such methods at a well-known point in the lifecycle, at which the app is known to be in its *running* state. These callbacks only depend on external events like incoming SMS messages, and not on the behavior of the application under analysis. Thus, they can occur at any time while the app is running.
These observations conveniently reduce the problem of modeling the framework to analyzing the effects of framework methods called by the application without impeding correctness or precision of the obtained analysis results. Section 5.1 describes in detail how FlowDroid generates sound dummy-main methods that respect callbacks in Android. A similar methodology must be followed for other client analyses or platforms other than Android.

6.6 Applying Summaries

After the summaries have been computed once, they can be used in an arbitrary number of taint analyses on client programs or Android apps. The library code is then no longer required. StubDroid integrates into FlowDroid using the concept of *Taint Wrappers* (see Section 4.9). Recall that taint wrappers are handlers for shortcut rules. They model external domain knowledge through an interface exposed by the taint-tracking engine. Whenever a method call is processed, the registered wrapper is asked whether it contains an explicit taint-propagation rule for the callee. StubDroid provides FlowDroid with a specialized taint wrapper implementation to inject the summary data during analysis. While this concept is specific to FlowDroid, other tools like CHEX [91] have similar extension points for explicit (library) method models and could thus use StubDroid's summaries in a similar fashion.
Assume the pair class from the motivating example in Listing 44 to be used in the program in Listing 49. This user code constructs an object of type `Data` and taints its field `b.c.d`. The data object is then passed to the `setComplex()` library method. This method is not part of the user code, but requires one to apply a library summary. Conceptually, the method copies the contents of the field `b.c` to `this.o1`. Therefore, the object containing the tainted data is returned by `get01()` and leaked in line 6. Note that in this example the actual tainted data is stored in a sub-field `d` which is never touched by the library implementation.
Recall the summary for the `setComplex()` method computed in Sec. 6.4: `this.o1.*` is tainted if `Parameter 0.b.c.*` is tainted. As explained in Section 6.2, the asterisk serves as a placeholder. When applying the summary rule to the example client code, StubDroid will therefore match the asterisk with the actual fields of the longer incoming access path and derive the more precise rule `this.o1.d.*` is tainted if `Parameter 0.b.c.d.*` is tainted.
This rule can then easily be matched against the incoming taint. FlowDroid will query the taint wrapper for the access path `data.b.c.d.*`. StubDroid matches the argument variable `data` against `Parameter 0` and then applies the rule. It reports `p.o1.d.*` back to the taint analysis. Note that the rules directly create new taints on access paths. If the client code direct read the field `p.o1` instead of calling a getter method, this would be captured by the taint on `p.o1.d.*`.

6.6.1 Aliasing

Library methods and user-code methods may both taint heap objects which may, in turn, have aliases both inside and outside of the library. The example in Listing 50 uses a `Pair` object to store a `Data` object. The code then retrieves this data object as `d1`, and taints one of its inner fields. Afterward, the same runtime object is retrieved

```
1 public void leakWithAliasing() {
2   Pair p = new Pair(new Data(), null);
3   Data d1 = (Data) p.get01();
4   d1.b.c = source();
5   Data d2 = (Data) p.get01();
6   leak(d2.b.c);
7 }
```

Listing 50: Library Summary Client with Aliases

again as `d2` before the data is read out and leaked. If the `Pair` class were not a library class but a part of the client code, FlowDroid could directly find the leak due to FlowDroid's built-in alias analysis (see Section 4.8). If we, however, assume that the `Pair.get01()` method is part of a library and its implementation is not available during the analysis of the client code, the library summaries must provide enough implementation for the alias analysis to work nevertheless.

At the moment, StubDroid is only compatible with FlowDroid's default flow-sensitive alias algorithm explained in Section 4.8.2. For the remainder of this section, we assume this algorithm to be used. Whenever a tainted value is assigned to a heap object, the data-flow engine automatically starts a backward tracking to find aliases. In the example, it would inter-procedurally propagate the access path `d1.b.c.*` backward to check whether this access path or some prefix of it is referenced on the right side of an assignment. All discovered aliases are then forward-propagated as first-class taints.

In Listing 50, however, the class `Pair` is abstracted away using a library summary. Therefore, FlowDroid cannot determine that `d1.b.c.*` aliases with `p.o1.b.c.*`. The relationship between `p.o1` and `d1` is encoded in `get01()`, but when the backward propagation reaches the call to `get01()`, there is no callee to process. Therefore, such aliasing relationships must also be encoded in the summaries. For this reason, taint wrappers in FlowDroid provide support for aliasing. Alias summaries in StubDroid are just like flow summaries, only with an inverse propagation rule. For `Pair.get01()`, there is already a rule: `return.*` is tainted if `this.o1.*` is tainted. Since the rule deals with heap objects, however, it can also be applied backward: If `return.*` is tainted afterward, `this.o1.*` may have been tainted before. In this direction, it encodes a may-alias relationship.

The final taint propagation in the example hence works as follows: When `p.get01()` is called first, nothing inside the `p` object is tainted yet, so the rule does not yet apply. When FlowDroid queries StubDroid's taint wrapper for aliases of `d1.b.c.*`, StubDroid can apply the inverse of the rule and return a taint on `p.o1.b.c.*` to FlowDroid. This taint is then propagated forward. In Line 5, the normal summary rule for `get01()` then applies and `d2.b.c.*` gets tainted. Therefore, FlowDroid can now detect the leak in Line 6.

To allow for more flexibility and some corner-cases (such as strings which are immutable), StubDroid stores a flag alongside every summary rule that indicates whether the flow may be inverted in order to answer alias queries.

6.6.2 Handling Incomplete Summaries

Note that taint wrappers can also be used with incomplete summaries. This is useful if libraries cannot be fully analyzed since, for instance, they depend on native code. In this case, analyzing the Java-based parts is still valuable, though it needs to be complemented with additional approximations. The FlowDroid client, for instance, supports two modes that determine how calls to methods are handled for which no summary is available. Refer to Section 4.9 for further information.

In the so-called *conservative mode*, the return value of a method call is always considered as tainted if the base object on which the method is invoked (or any field inside it) is tainted. In the example this would lead to a sound analysis even if the summary for the `getData()` method was missing, but may come at the cost of reduced precision. Similarly, the `hashCode()` and `equals()` methods can also be over-approximated with simple rules even if there is no summary for them for a certain class.

FlowDroid also supports the `exclusive` flag which allows a taint-wrapper implementation to claim the taints it generates for a specific call site and incoming taint as complete. If the wrapper declares itself as exclusive, the analysis will not consider the callee's implementation even if it is available. By default, the StubDroid taint wrapper is exclusive for all methods for which it has at least one summary fact in its input XML file, because this indicates that the respective class has been fully analyzed by StubDroid and thus all existing data flows have been summarized.

6.7 Evaluation

The summaries generated by StubDroid are maximally useful if they substantially reduce the time required to run the target analysis on a client program, if they do not reduce the precision of the analysis result (thus avoiding false positives), and if they are sound, i.e., do not introduce any false negatives. Our setup for computing the performance measures is explained in Section 6.7.1. The time required to compute a library summary is evaluated in section 6.7.2. Section 6.7.3 addresses the performance gains of using summaries and shows that StubDroid substantially reduces the time required for performing static analysis. In Section 6.7.4, we finally discuss the soundness and precision of our approach.

6.7.1 Experimental Setup

All performance experiments were carried out on a computation server featuring 40 virtual Xen CPU cores backed by Intel Xeon E5-4640 cores in physical hardware. The server was running Ubuntu 14.04 and Oracle's JVM version 1.7 in its default settings. Only the maximum heap size was set to 15 GB for summary generation and to 150 GB for analyzing apps. The large heap size for the app analysis was chosen to allow for a fair comparison with approaches that analyze the full Android library together with every app. Note that this may make some analyses perform less aggressive garbage collection than usual and thus report higher memory values than they would in more constrained scenarios.
To analyze the Android platform, we used an `android.jar` file manually built from a Galaxy Nexus device running Android 4.3. This is because the `android.jar` files included in the Android SDK as distributed by Google contain stub implementations only, which raise `NotImplementedException`s in every method. (They are only used to allow Android apps to link against the library interfaces.) As StubDroid is not only applicable to Android apps, but also to normal Java programs, we also evaluated it on the Java 8 (version 1.8.0_05) runtime library.
To evaluate the performance of StubDroid both in terms of summary application and summary generation, we computed summaries for the Android and the JDK implementations of the Java collections API. These classes are widely used in almost all applications, which is why modeling them is of high priority. Furthermore, these classes are rather large, and are thus suitable for assessing the scalability of the approach. The same holds for the string-processing classes, especially `java.lang.StringBuilder` which is used, e.g., to concatenate strings. With summaries for these two types of libraries, FlowDroid can successfully analyze most applications. All reported timings were averaged over 10 runs. The raw data is available on our project web page.

6.7.2 Summary Generation Performance

Generating the library summaries is a one-time effort that only needs to be repeated when the library is updated, which is rare in comparison to how often client applications using the library are analyzed. Nevertheless it is important that the summary generation is practically feasible. In Table 2, we report performance numbers on both the Android SDK and the Oracle JDK.
Our results show that StubDroid usually finishes in under three minutes per class for common Java collection APIs. For some of the concurrent collection implementations, the generation can take up to slightly over ten minutes. In these cases, the summaries generated by StubDroid are also considerably larger than those for their non-concurrent counterparts. This happens because the concurrent implementations need to assign additional internal synchronization fields which also become part of the summary and increase the complexity of the data flows in the respective methods.
Even though summaries could be centrally pre-computed on large servers, the memory requirements of StubDroid are modest. All summaries could be created within the 15 GB of heap space allotted, most of them using much less memory than available. This even applies to the large concurrent-collection classes such as the skip-lists.
Note that the Android and Oracle implementations of a particular class only share the specification, but not any source code. The versions shipped with the Android OS are especially optimized for resource-constrained devices. The differences are especially apparent in the number of flows generated for each class in Table 2. For `java.util.Priority Queue`, StubDroid detects more than twice as many flows in the JDK implementation than in the one from Android.

6.7.3 Analysis Performance

We next evaluate how StubDroid can impact the performance of a client taint analysis. We therefore ran FlowDroid on a number of Android test apps in three different modes:

Class	Generation Time (s)		Number of Flows		Memory (MB)	
	Oracle JDK	Android	Oracle JDK	Android	Oracle JDK	Android
java.util.ArrayDeque	45.82	52.76	82	88	1,086.52	2,203.27
java.util.ArrayList	37.95	42.52	72	72	1,086.52	2,323.40
java.util.HashMap	34.38	27.86	92	78	3,125.93	1,910.82
java.util.HashSet	51.46	45.04	140	81	3,125.93	1,910.82
java.util.LinkedHashMap	35.45	31.77	81	74	3,213.93	1,983.74
java.util.LinkedList	61.87	66.55	130	120	2,688.00	2,468.69
java.util.PriorityQueue	65.36	37.17	731	246	2,415.65	1,984.09
java.util.Stack	57.46	65.49	86	87	1,663.25	1,980.43
java.util.Vector	54.86	62.77	87	98	1,841.91	2,101.82
java.util.[...].ConcurrentHashMap	88.24	71.18	116	144	4,027.54	2,323.40
java.util.[...].ConcurrentLinkedQueue	64.23	35.58	36	40	1,509.77	1,761.86
java.util.[...].ConcurrentLinkedDeque	74.67	71.72	883	887	4,072.54	3,372.84
java.util.[...].ConcurrentSkipListMap	352.56	766.33	2206	2262	5,765.34	3,968.78
java.util.[...].ConcurrentSkipListSet	397.02	960.48	2234	3221	4,966.98	3,644.88
java.util.[...].DelayQueue	147.63	86.32	1745	807	4,864.34	3,106.15
java.lang.StringBuffer	92.06	81.73	397	309	1,212.16	2,621.19
java.lang.StringBuilder	86.35	81.72	443	305	1,617.32	2,425.11
Average	102.79	152.18	562	525	2,760.80	2,475.96

Table 2: Summary Generation Times for Android and JDK APIs

- Full analysis mode. In this mode, the full Android library was placed on the classpath and analyzed together with the app. Library summaries were not used.
- Hand-written summaries. In this mode, FlowDroid is run with its default hand-written summaries. This mode acts as a base-line as it corresponds to running the original FlowDroid implementation.
- StubDroid mode. In this mode, only library stubs were placed on the classpath and StubDroid's summaries were used to model the taint propagation over library call sites.

The hand-written summaries that FlowDroid uses by default not only cover the collection APIs but also a few other Android APIs such as cursors or intents. To allow for a fair comparison we generated StubDroid summaries for those APIs as well. The results in Table 3 show that StubDroid offers a performance that is comparable to FlowDroid's default hand-written summaries with a similarly high memory consumption; both analysis modes have comparable cost. In comparison to analyzing the full library implemenation together with every app, summaries provide a major decrease in runtime and memory consumption. In many cases, it even makes the analysis feasible; analyzing the app together with the full library implementation times out after 30 minutes.

The first two apps in Table 3 were taken from the DroidBench micro-benchmark suite, see Section 7. We chose those two apps of the suite that actually use library methods. These apps are rather small, so the discrepancy between the size of the app and the size of the library is significant. If no summaries are used, the analysis spends most of its time in the library. The other apps in the table are real-world applications taken from the Google Play Store.

Data-flow analysis is especially useful to find privacy violations in potentially malicious applications. Many malware apps steal the user's unique device identifier (IMEI), his phone number, or other personally-identifiable values. To assess how StubDroid can help with finding such data thefts, we assess how it impacts the performance of FlowDroid on 258 apps from four different malware families inside the well-known Malware Genome Project [157]. Table 4 shows the average runtimes for each malware family. For keeping the presentation concise, we only report values for the largest and most prevalent malware families. The column TO states the number of apps for which the analysis timed out after five minutes. In cases where all runs timed out, measurements are naturally not available (n/a). In cases where some runs timed out but not others, the numeric values indicate the average for the runs that did not time out.

The data in Table 4 indicates that analyzing the complete library together with every single app is infeasible and leads to timeouts in almost all cases. Using summaries, on the other hand, allows all but a handful of apps to be analyzed in under one minute. These time and memory savings, however, also depend on the precision of the summaries. For the *DroidDream Light* malware there are cases in which the hand-written summaries incur a higher

Application	Full		Hand-Written		StubDroid	
	Time (s)	Memory (MB)	Time (s)	Memory (MB)	Time (s)	Memory (MB)
ArrayAccess1	21.13	458.19	5.75	128.10	5.63	132.66
HashMapAccess1	21.37	493.88	5.99	173.70	5.96	174.75
Alipay	45.10	6,271.49	5.57	1,727.78	5.51	1,640.99
Avira Antivirus	Timeout	Timeout	48.72	3,908.80	38.18	2,662.60
Broncos News	Timeout	Timeout	4.90	1,571.53	4.86	1,373.82
Hamburg Casino	Timeout	Timeout	57.73	3,352.54	48.17	3,856.88
OpenTable	Timeout	Timeout	81.51	7,596.06	78.01	5,669.13
Wikipedia	46.87	3,884.95	1.48	270.01	1.59	445.81

Table 3: Summary Application Performance (Benign Applications)

		Full			Hand-Written			StubDroid		
Family	Apps	TO	Time (s)	Mem (MB)	TO	Time (s)	Mem (MB)	TO	Time (s)	Mem (MB)
ADRD	22	22	n/a	n/a	0	6.70	3,669.00	0	1.84	1,004.86
BaseBridge	121	115	25.03	1,210.95	7	17.53	525.03	0	6.54	311.30
DroidDream Light	46	45	63.30	1,222.58	0	7.41	7,309.21	0	5.26	451.32
Geinimi	69	69	n/a	n/a	0	29.79	849.36	1	6.91	281.52

Table 4: Summary Application Performance (Malware), TO = # of apps where analysis timed out

memory consumption than analyzing the full library. This is because the imprecise summaries lead to severe over-tainting during the analysis. The precise summaries computed by StubDroid do not show this memory explosion. Thus, StubDroid can considerably save time and memory in comparison to a full analysis as well as in comparision to the hand-written rule sets currently used by most static data-flow analyses.

Note that FlowDroid cannot complete the analysis of large applications when configured to analyze the full library implementation together with the app. In the very same hardware configuration, computing the summaries and then analyzing the app with these summaries does, however, complete in a reasonable amount of time. This is not a contradiction, because StubDroid analyzes every method inside the library in isolation and abstracts from its inner workings. Thus, it can create more concise access paths than FlowDroid's normal taint propagation and thereby drastically reduce the peak memory consumption of the data flow analysis.

Comparing the performance of the taint analysis with library models to the performance of a full analysis is not trivial since the summaries also cover methods that contain implicit flows (see Section 6.4.2). To allow the full analysis to also track taints over these paths, one would have to enable implicit-flow tracking for the complete target. This, however, is not the default in which apps are only scanned for explicit flows and only those few library methods that require implicit handling have models containing implicit flows. To circumvent this problem, we used the "full analysis" together with a simple hand-written taint wrapper for the data type conversion methods that would not faithfully propagate taints having a model of implicit flows.

6.7.4 Relative Soundness and Precision

A summary approach such as StubDroid can only be as precise and as sound as the analysis on which it is based. We thus rather check that StubDroid preserves analysis results, by comparing the data-flow results of two setups: (1) FlowDroid applied to apps including the complete library implementation and (2) FlowDroid applied to apps without the runtime library but with StubDroid's summaries instead. Ideally the same results should be achieved. Recall that StubDroid creates summaries that are applicable to arbitrary client programs. It therefore abstracts from concrete call sequences and states. It is important to evaluate whether this generalization leads to a loss of precision or soundness. We performed these comparisons on all apps from Tables 3 and 4 on which the full analysis terminated. We confirmed that all flows detected by FlowDroid that involved StubDroid-summaries were equal to a full analysis of the target app plus the library. This means that replacing the library implementation with StubDroid's summaries does not incur any penalty in precision or soundness. Note that this is also because we had

configured both StubDroid and FlowDroid with the same maximum access-path length of five. Naturally, precision might be reduced had we chosen less precise summaries.
In the Hamburg Casino app, FlowDroid's hand-written summary rules caused 15 false positives out of 36 flows in total (41,7%) due to a single overly aggressive rule. Another rule caused 1 false positive out of 3 flows (33,3%) in the Broncos News app. The StubDroid summaries avoided all of these false positives.

6.8 Related Work

In this Section, we compare StubDroid to existing approaches for summarizing library behavior (Section 6.8.1) and show a variety of existing work that can directly benefit from the summaries computed by StubDroid (Section 6.8.2).

6.8.1 Existing approaches for library summaries

The IFDS [117] framework, on which also FlowDroid is based, already computes low-level method summaries to improve efficiency if the same method is called multiple times in the same program. These summaries, however, are linked to the concrete context of the client analysis. Therefore, there is no easy way to serialize and re-use these summaries for multiple analysis runs, let alone different analyses. Naeem and Lhoták present a method for summarizing alias-analysis information [99], but no further data-flow relationships. StubDroid handles aliasing along with data flows. Aliases of objects that get tainted inside a library are modeled as first-class taints which avoids a separate summary concept. Aliasing relationships with objects in the client program must be computed anew for every client when applying the library summary, so no further summarization is possible.
Zuhu et al. [158] analyze implementations of clients to automatically infer library specifications. As they state, however, this requires an external oracle (e.g. a user consulting the documentation) to verify every generated specification candidate since the library code is not regarded at all. StubDroid on the other hand is fully automatic and directly analyzes the library implementations.
Rountev et al. [119] construct library summaries for IDE [120] data-flow analyses by first conducting a data-flow analysis on the library and then abstracting away redundant data-flow facts that are internal to the library. Rountev's approach, however, does not discuss how summaries can actually be abstracted in such a way that they can be persisted and can become useful for different clients. In fact, it appears that in their experiments the implementation stores and reuses summaries only within one and the same analysis process.
F4F [130] by Sridharan et al. is a system for performing taint analyses on framework-based web applications. It provides a specification language for modelling both the framework behavior and information from configuration files. Sample generators for such specifications are given for a number of web application frameworks. While F4F focuses on dispatch logic between web pages and accesses to user controls on them, StubDroid analyzes and summarizes data flows in basic framework methods and is fully automated. No specialized specification generator must be developed and maintained when the target framework changes.

6.8.2 Approaches benefiting from summaries

Library summaries are required for various analysis tools such as CHEX [91], which scans applications for potential cases of data misuse, e.g., when security vulnerabilities allow unauthorized access to an application's internal data. Apposcopy [49] detects Android malware based on semantic signatures describing data flows and inter-component communcations. This requires a precise flow detection which in turn needs precise library models. FlowDroid [9] is a data-flow tracker that can also analyze the library code together with the target application, but gains massive performance benefits from using StubDroid's summaries. AppSealer [155] is a tool for automatically patching component hijacking vulnerabilities in Android applications. It combines static and dynamic data-flow analyses to find vulnerable components to be patched. Library methods are handled using a coarse-grained default rule ("return value of method call is tainted if at least one parameter is tainted") with a few hand-written exceptions. Scandal [75] statically detects leaks of privacy-sensitive data in Android applications. The authors manually modeled the behavior of 220 commonly used Android framework methods to maintain precision. StubDroid automatically computes the library summaries required by these tools without the need for manual inspection of the library code. DroidSafe [60] requires the analyst to manually develop library stubs as Java code that are simplified versions of the original implementations. The authors have created 550 stub classes by hand which is a considerable effort that could be evaded by using StubDroid's data-flow summaries instead.

7 Benchmarking Analysis Tools: DroidBench

Assessing and comparing static (and also dynamic) data flow analysis tools is not trivial. Ideally, one wants to compare the tools along three different dimensions: precision, recall, and performance. In practice, memory consumption can also be a limiting factor when conducting a data flow analysis, but we will integrate this aspect into the performance dimension for the sake of simplicity. Clearly, comparing the performance of two tools is only meaningful if the precision and recall of the tool is compared as well. Otherwise, unfair tradeoffs can be made, i.e., missing many leaks for increased performance. For assessing precision and recall in data flow analysis, one needs a ground truth, i.e., a set of benchmark programs with a specified set of flows to be found. Real-world programs are not suitable as benchmarks, because reverse-engineering thousands of lines of code to obtain the expected set of data flows often requires a prohibitive manual effort. Note that the complete flow specification is required. Only checking the flows found by a specific tool can only indicate false positives. For finding missed leaks, the complete program code must be inspected manually. Therefore, comparative studies and evaluations require benchmarks that already come with a precise and complete data flow specification.
DroidBench aims at providing such a benchmark suite. It contains small Android apps that focus on particular challenges for data flow analysis tools such as storing sensitive data in a map and reading it back. DroidBench is not intended for performance testing, therefore the small app size is irrelevant for the purpose at hand. Most categories contain positive and negative challenges, i.e., apps that test recall and apps that test precision. In the current development version 3.0, DroidBench consists of 190 test cases in 18 categories. Most of the samples were created by us, but an increasing number of samples is also donated by other international research groups (approximately 85 as of version 3.0).
To ensure that the data-flow specification documented for every app matches the app's actual runtime behavior, all apps have been tested on actual smartphones. Neither FlowDroid nor any other tool we are aware of is able to achieve perfect recall and precision on DroidBench, nor would that be desirable. The goal of DroidBench is rather to offer challenges to the research community to further improve tools, approaches, and algorithms. We will first give an overview over the categories in DroidBench in Section 7.1 and then explain FlowDroid's performance on the test cases in Section 7.2.

7.1 Benchmark Categories

In this section, we will describe the 15 categories in DroidBench in more detail. For each category, we give examples of the challenges contained in them.

Aliasing

This category contains samples that test whether the analysis can correctly and precisely handle aliasing relationships. One test case, for instance, checks whether the alias analysis is flow-sensitive. Another test case contains a non-distributive alias problem.

Android-Specific Challenges

The focus of this category is on how well an analysis handles the Android framework. Test cases address calls to library methods inside the Android SDK and configuration options in the `AndroidManifest.xml` file. Other tests check whether Android's serialization mechanism for inter-component communication (parceling) is modeled correctly. Note that in this category, no actual inter-component communication is performed, though. In this category, Android's possibility of nesting user-defined libraries in apps is tested as well.

Arrays and Lists

The test apps in this category use arrays, lists, and maps to temporarily store data. The data is then read back and passed to a sink method. The positive tests read back actual sensitive data, while the negative tests read constant strings from the same arrays, lists, and maps that also hold sensitive data. An imprecise analysis would cause a false positive here if it cannot distinguish the different elements inside the data structure.

Callbacks

These tests address callbacks in Android. Some callbacks are registered in code, others in the layout XML files. Some callbacks are kept perpetually, others are deregistered at some point in the app's lifecycle to check whether an analysis tool correctly handles the temporal properties of Android callbacks. The tests for callback chains (one call-

back registers another one) are also part of this category, as well as tests on callback ordering and the relationship between callbacks and the component lifecycle.

Dynamic Code Loading
The test cases in this category dynamically load additional code that is then called via reflection. A static analysis tool must include this code as well as the "normal" app code, or else would miss the leaks. Depending on the test case, the source, the sink, or intermediate parts of the data flow (or any combination thereof) are hidden inside the dynamically-loaded code.

Emulator Detection
This category mainly addresses challenges for dynamic analysis tools. The tests try to detect whether they are run inside an emulated environment. If so, no data leakage is performed. Dynamic analysis tools must make sure to very precisely model an actual device to find these leaks. For static analysis tools, these tests are usually trivial as the flow itself simply directly passes the sensitive data into a sink.

Field and Object Sensitivity
These tests can be used to check whether a data flow analysis tool can correctly distinguish different fields in the same base object and different base objects for the same tainted field. Test for flow-sensitivity are also contained in this category, i.e, passing field data to the sink method before actually tainting the field.

General Java
This category contains test cases for standard Java language constructs including exceptions, loops, static initializers and fields, string manipulation, and virtual dispatch. None of these test cases are specific to Android or any other framework. An imprecise analysis would cause false positives here by over-approximating potential callees for a given call site, or would miss leaks by not propagating taint across exceptional control flow edges.

Implicit Flows
The implicit flow handling of the data flow analysis tool is tested in this category. Sensitive data is re-formatted using conversion maps, or is checked and a single bit is leaked depending on whether the sensitive data is equal to a given constant value.

Inter-App Communication
In this category, test cases do not consist of single apps, but of multiple apps that collude to conduct a data leakage. Data is obtained in one app, and sent on to a different app that relays it to a third app where the data is finally leaked. To correctly detect the leaks in this category, a data flow analysis must be able to map Android intents to the correct receiver app and extend the data flow paths over the boundaries of multiple apps.

Inter-Component Communication
The test cases inside this category are similar to the inter-app test cases. The key difference is that they transfer sensitive data between different components inside the same app instead of between components spread across multiple apps. There are test cases for one activity starting another one while passing tainted data, for sending intents between components, and for exchanging data between components through Android's `SharedPreference` mechanism. These test cases also use various techniques for identifying the receiver of an intent (explicit and implicit intents, the latter through various matching mechanisms such as action strings).

Lifecycle
How data flow analysis tools handle the Android lifecycle is tested with the test cases in this category. These apps obtain and leak data in various lifecycle stages. Only a tool that correctly models the complete lifecycle of all the four different Android components plus fragments and Android application objects can achieve full precision and recall in these tests. If the ordering of lifecycle events is over-approximated, false positives will occur. Missing component types or lifecycle methods will result is missed leaks. Note that these test cases not only rely on the well-documented lifecycle methods from the overview chart (see Figure 14 for the Activity component), but also uses other less known methods provided by the Android SDK.

Native Code
While the test cases in all other DroidBench categories (except for the *Self-Modification* category) are purely implemented in Java, the test cases in this category use the Java Native Interface (JNI) to call native methods written in C/C++. The tests cover four scenarios: (1) Sensitive data is obtained in Java code, passed through native code, and leaked back in the Java code, (2) the call to the source is already in native code, but the sink is still in Java code, (3) the source is in Java code, but the sink is in native code, and (4) both source and sink are in native code.

Reflection
The test cases in this category use reflection to call methods and access fields rather than directly invoking or accessing them. Some tests also further obfuscate the strings that identify the targets of the reflective method or field accesses.

Reflection_ICC
These test cases combine reflective method calls with inter-component communication. They emulate obfuscation techniques used in modern malware apps to prevent static analysis tools from identifying the contents and targets of intents. The strings used in the reflective calls are also encoded or dynamically constructed in some of the tests to make them unavailable to tools that only scan for constant strings.

Self-Modification
This category contains apps that modify their own code at runtime. The apps contain native code that is invoked directly after the main activity's class is loaded into the Dalvik runtime. This native code then locates the Dalvik code in memory and modifies the targets of method calls so that previously innocent call sequences now leak sensitive information. If the static analysis only considers the original app code without these runtime modifications, it misses the leak.

Threading
Java offers various APIs for creating new threads. The Android operating system adds more techniques to execute asynchronous tasks. The test cases in this category check whether an analysis tool correctly handles code that executes outside of the main thread, but potentially exchanges data with the main thread or other threads in general.

Unreachable Code
These test cases contain data flows that are not actually executed at runtime, because the data flow path (or at least parts of it) are unreachable. The challenge for a static analysis tool is to identify this unreachable code. For not generating false positives, the analysis must be able to construct constraints on the outcomes of arithmetic operations and check the compatibility of conditionals against these constraints. If the test case, for instance, creates a random value in the range from zero to 100 and then checks whether the value is larger than 110, the analysis must be able to derive that this will never hold at runtime.

7.2 Evaluating FlowDroid on DroidBench

In this Section, we report on the precision and recall of FlowDroid on the various benchmarks in DroidBench. For the test cases on which FlowDroid fails, we give reasons. Table 6 shows all test cases in DroidBench and compare's FlowDroid's output to the expected results as well as other commercial and academic, state-of-the-art static data flow analysis tools. Since the tools differ in their scope, we first explain our setup and the configurations we use for the tools.

7.2.1 FlowDroid Configuration

FlowDroid does not provide support for inter-component and inter-app communication. To evaluate the tool in a meaningful way, we configure all incoming intents as sources. This configuration over-approximates the actual taint problem by considering all incoming intents as tainted, regardless of where they come from and what their actual state is. The naturally leads to false positives in cases in which a component never receives an intent or the incoming intent does not contain any tainted data. Such false positives are counted as false positives, because they do not reflect any real leaks. On the other hand, if FlowDroid detects a leak from an incoming taint to a sink method and this is actually part of a real leaking path, we count it as a hit, even though FlowDroid does not

Setting	Value	Explanation
Access Path Length	5	See Section 3.3
Symbolic Access Paths	Enabled	Section 3.3
Path Agnostic Results	Enabled	Section 4.13
Data Type Propagation	Enabled	Section 4.11
Implicit Flow Tracking	Disabled	Section 4.6
Track Static Fields	Enabled	Section 4.2
Track Exceptional Flows	Enabled	Section 4.5
Array Size Tainting	Disabled	Section 4.4
Alias Algorithm	Flow-Sensitive	Section 4.8
Flow Sensitive Aliasing	Enabled	Section 4.8.4
Callgraph Algorithm	SPARK	Section 4.2.6
Code Optimization	Constant Prop.	Section 4.14
Android Callbacks	Enabled	Section 5.3
Callback Sources	Enabled	Section 5.3.7
Callback Analyzer	Iterative	Section 5.3.4
Layout Matching	Sensitive Only	Section 5.2

Table 5: Configuration for FlowDroid Evaluation on DroidBench

detect the real source in the sender component. In that case, FlowDroid technically discovered only a fraction of the taint path and reported a "wrong" source, because the original source was not the incoming intent, but, e.g., the call to `getDeviceId()` in some other component. Still, we consider this as a true leak for the purpose of our evaluation. Furthermore, if one component sends sensitive data to another component using an intent, this will always be reported as a leak. FlowDroid has no mechanism to check whether the data actually leaves the current app and might be observed by a third party, or in which component the data flow continues. We count such reports as correct hits according to our source/sink specification. These hits, though semantically incorrect, are enough for tools such as ICCTA to build upon FlowDroid and add more precise intent matching.
For library handling, we rely on FlowDroid's built-in *Easy Taint Wrapper* to allow for a more fair comparison to other tools which do not provide sophisticated library handling. We find that this decision does not lead to any false positives in the DroidBench suite, but to a small number of false negatives. These false negatives are also shared by other tools. In total, we used FlowDroid with all default settings as shown in Table 5.

7.2.2 IBM AppScan Source Configuration

For our evaluation, we used IBM AppScan Source in version 9.0.3. We ran the analysis client on Windows 10 on a physical machine. The server-side components (both the IBM AppScan Enterprise Server and the IBM AppScan Source Server) were installed in a VM running CentOS 7 hosted on the same system. We chose CentOS, because it is a free binary-compatible alternative to RedHat Enterprise Linux which is the only officially supported Linux platform for AppScan Source. We ran the tool on the source code of the DroidBench test cases, because it does not support binary analysis. Unfortunately, the default source and sink list of AppScan Source is not available, so we added missing sources and sinks to the custom rule list on demand (i.e., when we noticed missing flows that were potentially due to unknown sources or sinks). Due to the low number of distinct sources and sinks in the DroidBench test cases, this approach was feasible. For taint propagators (a concept similar to FlowDroid's EasyTaintWrapper explained in Section 4.9.1), we used the approach of same on-demand extensions to the database.
There are two different types of findings in AppScan Source: *Security Findings*, which are in turn classified as either *Definite* or as *Suspect*, and *Scan Coverage Findings*. Only security findings are linked to data flows. Scan coverage findings can be reported only because a specific API call is present in the code or a specific pattern matches. Consequently, including these findings into the result set increases the risk of false positives. For a fair comparison, we include both results (without and with scan coverage findings) in our result table. The combination of both types of findings is shown in the second column *AppScan Ext* (for "extended"). Note that these findings are always in addition to the trace-based findings, so they can reduce the number of false negatives, but cannot improve on false positives. To allow for a fair comparison, we never counted security-related findings that were obviously for

other reasons than the information flow to be found in the respective app, e.g., a stack trace from an exception being potentially leaked to an attacker. Since AppScan is a generic security assessment tool and not only a pure data flow tracker, it also reports various findings of that kind.

7.2.3 DroidSafe Configuration

We also tried to compare the precision and recall of FlowDroid to DroidSafe [60]. However, we found that the DroidSafe analysis times out for some of the DroidBench test cases. We explicitly report such cases and only compare the outputs of the tool for such cases in which the analysis finished within 10 minutes. Given that all DroidBench micro-benchmark apps are smaller than one megabyte and only contain a few lines of user code, we can require a tool to finish the analysis within such a timeframe. FlowDroid finishes on the DroidBench examples well in under one minute per app, requiring only a handful of seconds oftentimes. For DroidSafe, even when the analysis terminates, it still requires multiple minutes of processing time, making the scalability of the tool to real-world apps questionable. We used DroidSafe with the default configuration on APK files, not source code. For some apps, DroidBench failed with an exception, which we also explicitly denoted.
DroidSafe supports inter-compoent data flow tracking. Therefore, for some of the apps from the inter-component communication category inside DroidBench, it not only finds the intra-component flows that end at methods such as `startActivity()`, but correctly links the outgoing data to its use in the receiver component. Since the other tools in the comparison (especially FlowDroid and AppScan Source) do not support inter-component analysis, we count the leak as found if DroidSafe either found the flow to the component boundary or to the use in the correct receiver component. We only count a false negative if the expected flow was not found at all. If DroidSafe reports a flow to a wrong receiver component (i.e., made a wrong intent mapping), we count this as a false positive. The results of the other tools are a counted in the same way, i.e., FlowDroid also receives false positives for reporting flows inside components that are never triggered, though it was never designed to analyze inter-component communication.

7.2.4 JoDroid Configuration

We ran JoDroid on the DroidBench test cases according to the description on their Github wiki page[32]. The process consists of three steps. Firstly, the app's manifest file and code is analyzed to find the entry points for the later analysis steps. We used the *normal* scan mode which is also the default in the examples given in the documentation. This step yields a `.ntrP` file. Afterwards, JoDroid constructs the system dependence graph (SDG) of the app and saves it into the `.pdg` file. We used the option *–analysis full* to construct an object-sensitive SDG, and the option `-construct all` to consider all components that can receive external intents as possible starting points for the analysis, not only the activities that appear in the device's app launcher[33]. Afterwards, we use the Joana GUI to read in the constructed `pdg` file, configure the sources and sinks, and run the actual data flow analysis. In some cases, the analysis timed out in one of the steps explained above, or the respective programs terminated with an unhandled exception. We denote those cases explicitly in the result table.
One challenge we faced with JoDroid is that we could not globally configure the sources and sinks. Instead of definiting a method such as `getDeviceId()` as a source and have the tool automatically match this definition to all calls to this method, we had to manually find all calls to the methods we were interested in and manually define each of these calls as sources or sinks, respectively. The reason for this required extra effort is that JoDroid analyzes the app's code for method invocations. The `getDeviceId()` method is called on a base object that is retrieved from a factory method implemented in the Android framework. Since the Android SDK is shipped with stub versions of the framework methods, no object is actually returned by these factory methods from the perspective of the analyzer. Therefore, since the base object is not available, the analyzer does not consider the call to `getDeviceId()` to have any outgoing call edges, and, consequently, the `getDeviceId()` method is never called. Source definitions on that method are therefore never applied. Directly selecting the individual call sites removes the dependency on this part of the callgraph and allows the analysis to be configured correctly nevertheless.

7.2.5 Results Table

The following table 6 shows an overview over the results of the various tools on the DroidBench test cases. In the cases in which a tool did not complete the analysis of an app due to an uncaught exception, we denote *Exception*

[32] `https://github.com/joana-team/joana/tree/master/wala/joana.wala.jodroid`
[33] Those activities that appear in the launcher are sometimes referred to as "main" activities in the literature.

instead of the test results. If the analysis timed out, we denote *Timeout*. For calculating the metrics *precision*, *recall*, and *F-Measure*, we treat such cases as if the tool had returned zero results. Note that this treatment has the implicit consequence that the metrics lean towards high precision: A tool that fails contributes zero false positives to the metric. If there were no leak to be found in the respective DroidBench app, the crash is thus equal to a fully correct result.

Each DroidBench test case contains a structured definition of the test case and the leaks to be found. This information is given as a comment in the header of the main Java source code file. In some cases, the number of expected leaks defined in the source code of a test case differs from the expected findings in Table 6. We made these changes to allow for a fair comparison between the different tools. The DroidBench test cases that, for instance, leak the user's location, i.e., both the longitude and the latitude of the current GPS position, are defined with two expected leaks in the respective source code files. Most tools, however, consider this to be only a single leak, because the same complex object, i.e., the location, is affected, regardless of how many of its properties are leaked. We therefore counted these test cases to have only one leak in the table. If a tool detects either the location leak as a single atomic leak, or the two individual properties as two separate leaks, it is assumed to be fully correct on the test case and produce the one expected leak for the table.

⊛ = correct warning, ⋆ = false warning, ○ = missed leak
multiple circles in one row: multiple leaks expected
all-empty row: no leaks expected, none reported

App Name	AppScan	AppScan Ext	DroidSafe	JoDroid	FlowDroid
Aliasing					
FlowSensitivity1	⋆	⋆	Timeout	⋆	
Merge1		⋆	Timeout	⋆	⋆
SimpleAliasing1	⊛	⊛	Timeout	⊛	⊛
StrongUpdate1	⋆	⋆	Timeout	⋆	
Arrays and Lists					
ArrayAccess1	⋆	⋆	⋆	⋆	⋆
ArrayAccess2	⋆	⋆	⋆	⋆	⋆
ArrayAccess3	○	⊛	⊛	⊛	⊛
ArrayAccess4		⋆			
ArrayAccess5		⋆			
ArrayCopy1	⊛	⊛	⊛	○	⊛
ArrayToString1	⊛	⊛	⊛	○	⊛
HashMapAccess1	⋆	⋆	⋆	⋆	⋆
ListAccess1	⋆	⋆	⋆	⋆	⋆
MultidimensionalArray1	⊛	⊛	⊛	⊛	⊛
Callbacks					
AnonymousClass1	○	⊛	⊛	⊛	⊛
Button1	○	⊛	⊛	⊛	⊛
Button2	⊛ ○ ○	⊛ ⊛ ⊛ ⋆	⊛ ⊛ ⊛ ⋆	⊛ ⊛ ⊛ ⋆	⊛ ⊛ ⊛ ⋆
Button3	○ ○	⊛ ⊛	⊛ ○	⊛ ⊛	⊛ ⊛
Button4	○	⊛	⊛	⊛	⊛
Button5	○	⊛	⊛	○	⊛
LocationLeak1	○ ○	⊛ ⊛	⊛ ⊛	⊛ ⊛	⊛ ⊛
LocationLeak2	○ ○	⊛ ⊛	⊛ ⊛	⊛ ⊛	⊛ ⊛
LocationLeak3	○	⊛	⊛	⊛	⊛
MethodOverride1	⊛	⊛	⊛	⊛	⊛
MultiHandlers1		⋆ ⋆			
Ordering1		⋆ ⋆	⋆ ⋆		
RegisterGlobal1	○	⊛	⊛	○	⊛
RegisterGlobal2	○	⊛	⊛	○	⊛
Unregister1	⋆	⋆	⋆		⋆
Dynamic Code Loading					
DynamicBoth1	○	⊛ ⋆	○	○	○
DynamicSink1	○	⊛ ⋆	○	○	○

⊛ = correct warning, ⋆ = false warning, ○ = missed leak
multiple circles in one row: multiple leaks expected
all-empty row: no leaks expected, none reported

App Name	AppScan	AppScan Ext	DroidSafe	JoDroid	FlowDroid
DynamicSource1	○	⊛ ⋆ ⋆	○	○	○
Emulator Detection					
Battery1	⊛	⊛	⊛	⊛	⊛
Bluetooth1	⊛	⊛	⊛	⊛	⊛
Build1	⊛	⊛	⊛	⊛	⊛
Contacts1	⊛	⊛	⊛	⊛	⊛
ContentProvider1	⊛ ⊛	⊛ ⊛	Timeout	⊛ ⊛	⊛ ⊛
DeviceId1	⊛	⊛	⊛	⊛	⊛
File1	⊛	⊛	⊛	⊛	⊛
IMEI1	○ ○	⊛ ⊛	○ ○	⊛ ⊛	⊛ ⊛
IP1	⊛	⊛	⊛	⊛	⊛
PI1	⊛	⊛	⊛	⊛	⊛
PlayStore1	⊛ ⊛	⊛ ⊛	Timeout	⊛ ⊛	⊛ ⊛
PlayStore2	⊛	⊛	Timeout	Timeout	⊛
Sensors1	⊛	⊛	⊛	⊛	⊛
SubscriberId1	⊛	⊛	⊛	⊛	⊛
VoiceMail1	⊛	⊛	⊛	⊛	⊛
Field and Object Sensitivity					
FieldSensitivity1		⋆			
FieldSensitivity2		⋆			
FieldSensitivity3	○	⊛	⊛	⊛	⊛
FieldSensitivity4	⋆	⋆			
InheritedObjects1	○	⊛	⊛	⊛	⊛
ObjectSensitivity1		⋆		⋆	
ObjectSensitivity2	⋆	⋆	⋆		
Implicit Flows					
ImplicitFlow1	○	⊛	⊛	⊛	⊛
ImplicitFlow2	○ ○	○ ○	○ ○	⊛ ⊛	⊛ ⊛
ImplicitFlow3	○ ○	○ ○	○ ○	⊛ ⊛	⊛ ⊛
ImplicitFlow4	○ ○	○ ○	○ ○	⊛ ⊛	⊛ ⊛
ImplicitFlow5	○	○	○	○	⊛
ImplicitFlow6					
Inter-App Communication					
Echoer	⊛ ⊛ ○	⊛ ⊛ ○	⊛ ⊛ ⊛	⊛ ⊛ ⊛	⊛ ⊛ ⊛
SendSMS	⊛ ⊛ ⊛	⊛ ⊛ ⊛	⊛ ⊛ ⊛	Exception	⊛ ⊛ ⊛
StartActivityForResult1	⊛ ⊛ ⊛ ⊛ ⊛ ⊛	⊛ ⊛ ⊛ ⊛ ⊛ ⊛	⊛ ⊛ ⊛ ⊛ ⊛ ⊛	Exception	⊛ ⊛ ⊛ ⊛ ⊛ ⊛
Collector	○ ○	○ ○	⊛ ○	⊛ ⊛	⊛ ⊛
DeviceId_Broadcast1	⊛	⊛	Exception	○	⊛
DeviceId_ContentProvider1	⊛ ○	⊛ ○	Exception	Exception	⊛ ⊛
DeviceId_OrderedIntent1	⊛	⊛	Exception	⊛	⊛
DeviceId_Service1	⊛	⊛	Exception	○	⊛
Location1	○ ○	○ ○	Exception	Exception	⊛ ⊛
Location_Broadcast1	○ ○	○ ○	Exception	○ ○	⊛ ⊛
Location_Service1	○ ○	○ ○	⊛ ⊛	○ ○	⊛ ⊛
Inter-Component Communication					
ActivityCommunication1	○	⊛	⊛	○	⊛
ActivityCommunication2	○ ⋆	⊛ ⋆	⊛	⊛ ⋆	⊛ ⋆
ActivityCommunication3	○ ⋆	○ ⋆	⊛	⊛ ⋆	⊛ ⋆
ActivityCommunication4	○ ⋆	○ ⋆	⊛	⊛ ⋆	⊛ ⋆
ActivityCommunication5	○ ⋆	○ ⋆	⊛	⊛ ⋆	⊛ ⋆

⊛ = correct warning, ⋆ = false warning, ○ = missed leak
multiple circles in one row: multiple leaks expected
all-empty row: no leaks expected, none reported

App Name	AppScan	AppScan Ext	DroidSafe	JoDroid	FlowDroid
ActivityCommunication6	○ ⋆	○ ⋆	⊛	⊛ ⋆	⊛ ⋆
ActivityCommunication7	○ ⋆	○ ⋆	⊛	⊛ ⋆	⊛ ⋆
ActivityCommunication8	○ ⋆	○ ⋆	⊛	⊛ ⋆	⊛ ⋆
BroadcastTaintAndLeak1	⊛	⊛	⊛	⊛	⊛
ComponentNotInManifest1	⋆	⋆		⋆	⋆
EventOrdering1	○ ⋆	○ ⋆	⊛	○ ⋆	○ ⋆
IntentSink1	○	○	⊛	⊛	⊛
IntentSink2	⊛	⊛	⊛	Exception	⊛
IntentSource1	⊛ ⊛	⊛ ⊛	⊛ ⊛	Exception	⊛ ⊛
ServiceCommunication1	○	○	⊛	Exception	○
SharedPreferences1	○	○	⊛	○	○
Singletons1	○	○	⊛	⊛	○
UnresolvableIntent1	⊛ ○	⊛ ○	⊛ ⊛	⊛ ⊛	⊛ ⊛
Lifecycle					
ActivityEventSequence1	○	⊛	⊛	⊛	⊛
ActivityEventSequence2					(see notes)
ActivityEventSequence3	○	⊛	⊛	⊛	⊛
ActivityLifecycle1	○	⊛	⊛	○	⊛
ActivityLifecycle2	○	⊛	⊛	⊛	⊛
ActivityLifecycle3	○	○	⊛	⊛	⊛
ActivityLifecycle4	○	⊛	⊛	⊛	⊛
ActivitySavedState1	○	○	⊛	○	⊛
ApplicationLifecycle1	○	⊛	⊛	⊛	⊛
ApplicationLifecycle2	○	⊛	⊛	○	⊛
ApplicationLifecycle3	○	⊛	⊛	⊛	⊛
AsynchronousEventOrdering1	○	⊛	⊛	⊛	⊛
BroadcastReceiverLifecycle1	○	⊛	⊛	⊛	⊛
BroadcastReceiverLifecycle2	○	⊛	⊛	○	○
BroadcastReceiverLifecycle3	⊛	⊛	⊛	⊛	⊛
EventOrdering1	⊛	⊛	⊛	⊛	⊛
FragmentLifecycle1	○	⊛	Timeout	⊛	○
FragmentLifecycle2	○	○	⊛	Timeout	○
ServiceEventSequence1	○	⊛	⊛	○	○
ServiceEventSequence2	○	⊛	⊛	Exception	○
ServiceEventSequence3	○	⊛	⊛	Exception	⊛
ServiceLifecycle1	○	⊛	⊛	⊛	⊛
ServiceLifecycle2	⊛	⊛	⊛	○	⊛
SharedPreferenceChanged1	⊛	⊛	⊛	○	⊛
General Java					
Clone1	⊛	⊛	⊛	⊛	⊛
Exceptions1	⊛	⊛	⊛	⊛	⊛
Exceptions2	⊛	⊛	⊛	⊛	⊛
Exceptions3	⋆	⋆	⋆	⋆	⋆
Exceptions4	⊛	⊛	⊛	○	⊛
Exceptions5	⊛	⊛	⊛	○	⊛
Exceptions6	⊛	⊛	⊛	○	⊛
Exceptions7	⋆	⋆	⋆		
FactoryMethods1	⊛ ⊛	⊛ ⊛	⊛ ⊛	⊛ ⊛	⊛ ⊛
Loop1	⊛	⊛	⊛	⊛	⊛
Loop2	⊛	⊛	⊛	⊛	⊛
Serialization1	○	○	⊛	○	○

⊛ = correct warning, ⋆ = false warning, ○ = missed leak
multiple circles in one row: multiple leaks expected
all-empty row: no leaks expected, none reported

App Name	AppScan	AppScan Ext	DroidSafe	JoDroid	FlowDroid
SourceCodeSpecific1	⊛	⊛	⊛	⊛	⊛
StartProcessWithSecret1	⊛	⊛	⊛	○	⊛
StaticInitialization1	○	⊛	⊛	○	○
StaticInitialization2	○	⊛	⊛	⊛	⊛
StaticInitialization3	○	⊛	⊛	○	○
StringFormatter1	⊛	⊛	⊛	○	○
StringPatternMatching1	⊛	⊛	⊛	⊛	⊛
StringToCharArray1	⊛	⊛	⊛	⊛	⊛
StringToOutputStream1	⊛	⊛	⊛	⊛	⊛
UnreachableCode	⋆	⋆		⋆	
VirtualDispatch1	○	⊛ ⋆	⊛	⊛	⊛ ⋆
VirtualDispatch2	○	⊛ ⋆	⊛	⊛	⊛ ⋆
VirtualDispatch3		⋆	Timeout		⋆
VirtualDispatch4		⋆	Timeout		
Miscellaneous Android-Specific					
ApplicationModeling1	○	⊛	⊛	○	⊛ [34]
DirectLeak1	⊛	⊛	⊛	⊛	⊛
InactiveActivity	⋆	⋆		⋆	
Library2	○	⊛	⊛	⊛	⊛
LogNoLeak		⋆			
Obfuscation1	⊛	⊛	○	⊛	⊛
Parcel1	○	⊛	⊛	○	⊛
PrivateDataLeak1	○	⊛	⊛	⊛	⊛
PrivateDataLeak2	⊛	⊛	⊛	⊛	⊛
PrivateDataLeak3	⊛	⊛	⊛	○	○
PublicAPIField1	⊛	⊛	⊛	○	⊛
PublicAPIField2	⊛	⊛	⊛	⊛	⊛
View1	○	⊛	⊛	⊛	⊛
Native Code					
JavaIDFunction	○	○	○	○	○
NativeIDFunction	○	○	○	○	○
SinkInNativeCode	○	○	○	○	○
SinkInNativeLibCode	○	○	○	○	○
SourceInNativeCode	○	○	○	○	○
Reflection					
Reflection1	⊛	⊛	⊛	⊛	⊛
Reflection2	⊛	⊛	⊛	○	⊛
Reflection3	○	⊛	⊛	○	⊛
Reflection4	⊛	⊛	⊛	○	⊛
Reflection5	○	○	○	○	⊛
Reflection6	○	○	⊛	○	⊛
Reflection7	○	○	○	○	○
Reflection8	○	○	○	○	⊛
Reflection9	○	○	○	○	⊛
Reflection_ICC					
ActivityCommunication2	○ ⋆	○ ⋆	○	○	⊛ ⋆
AllReflection	○	○	○	○	○
OnlyIntent	○	○	○	○	⊛

[34] A model for the `getApplication()` method in `android.app.Activity` was contributed to the FlowDroid open source project by Gal Dudovitch and Daniela Rabkin from Tel-Aviv University.

⊛ = correct warning, ⋆ = false warning, ○ = missed leak
multiple circles in one row: multiple leaks expected
all-empty row: no leaks expected, none reported

App Name	AppScan	AppScan Ext	DroidSafe	JoDroid	FlowDroid
OnlyIntentReceive	○	○	○	○	○
OnlySMS	○	○	○	○	⊛
OnlyTelephony	○	○	⊛	○	⊛
OnlyTelephony_Dynamic	○ ○	○ ○	⊛ ⊛	○ ○	⊛ ⊛
OnlyTelephony_Reverse	○	○	⊛	○	⊛
OnlyTelephony_Substring	○	○	⊛	○	⊛
SharedPreferences1	○	○	○	○	○
Self-Modification					
BytecodeTamper1	○	⊛	○	○	○
BytecodeTamper2	○	○	○	○	○
BytecodeTamper3	○	⊛	○	○	○
BytecodeTamper4	○	○	○	○	○
Threading					
AsyncTask1	○	⊛	⊛	○	⊛
Executor1	○	⊛	⊛	○	⊛
JavaThread1	○	⊛	⊛	○	⊛
JavaThread2	○	○	⊛	○	⊛
Looper1	○	○	Timeout	○	⊛ [35]
TimerTask1	○	⊛	Timeout	⊛	⊛
Unreachable Code					
SimpleUnreachable1			Exception	Timeout	
UnreachableBoth1			⋆	⋆	⋆
UnreachableSink1			⋆	⋆	⋆
UnreachableSource1			⋆	⋆	⋆
Sum, Precision and Recall					
⊛, higher is better	70	131	136	97	158
⋆, lower is better	23	28	14	24	24
○, lower is better	118	57	34	70	30
Additonal ○ (Crashes / Timeouts)	-	-	18	21	-
Precision $p = ⊛/(⊛ + ⋆)$	75.27%	82.39%	90.67%	80.17%	86.81%
Recall $r = ⊛/(⊛ + ○)$	37.23%	69.68%	72.34%	51.60%	84.04%
F-measure $2pr/(p+r)$	0.50	0.76	0.80	0.63	0.86
Stability					
Apps with Crashes / Timeouts	-	-	19	12	-

Table 6: DroidBench test results

7.3 FlowDroid Result Explanation

In this Section, we discuss in detail the DroidBench test cases on which FlowDroid fails, i.e., produces at least one false positive or false negative.

[35] A model for the `android.os.Handler` class was contributed to the FlowDroid open source project by Gal Dudovitch and Daniela Rabkin from Tel-Aviv University.

7.3.1 Aliasing

Merge1
This test case contains a non-distributive aliasing problem. Since FlowDroid over-approximates the set of possible aliases for a given taint abstraction, a wrong alias is found, leading to a false positive. See Section 4.8.4 for further details on this issue.

7.3.2 Arrays and Lists

ArrayAccess1 & 2
This test case writes data into an array, but then reads back non-sensitive from a different index in the same array. Since FlowDroid over-approximates the taint state of arrays, the whole array gets tainted and the false positive occurs.

HashMapAccess1
In this test case, sensitive data is written into a hash map. The test case then reads back non-sensitive data for a different key in the same hash map and leaks it. FlowDroid over-approximates the taint state of the map and assumes that all data read from the map is tainted if the map holds at least one tainted item.

ListAccess1
Similar to `HashMapAccess1`. FlowDroid over-approximates the taint state of lists just like it does with maps. In this test case, sensitive data is stored in a list, but the data that gets leaked is non-sensitive read from a different index in the same list.

7.3.3 Callbacks

Button2
In this test case, a global field holding sensitive data is overwritten with an empty string before passing this value to the sink method. Though FlowDroid supports strong updates, it only does so intra-procedurally. In this case, it cannot kill the taint as it does not know that both accesses to the same field must always alias, i.e., a real strong update would be possible.

Unregister1
This test case registers a callback handler that leaks data, but unregisters it again before it can actually be triggered. Unregistering callback handlers is not supported in FlowDroid, we assume that all callbacks are active eternally while their host component is running. See Section 5.3 for details.

7.3.4 Dynamic Code Loading

FlowDroid only analyzes the code contained in the main `classes.dex` file of the APK. It does not consider additional code loaded at runtime. Therefore, false negatives occur if the sources and/or sinks are contained in dynamically-loaded code or if the data flow passs through such code unavailable to the analysis.

7.3.5 Inter-App Communication

FlowDroid is a tool for tracking intra-component data flows. We therefore analyze each app from the inter-app benchmark set individually. If the leaks from the app's boundaries (e.g., incoming intents or data obtained from the Android operating system) to the boundaries (e.g., outgoing intents or external resources such as files) are found, the respective leaks are counted. We do not assess combinations of apps as FlowDroid does not match intent senders and receivers.

7.3.6 Inter-Component Communication

ActivityCommunication2-8
FlowDroid over-approximates the inter-component data flows. Every incoming intent is treated as a source. Therefore, when an activity sends an intent to another one and this second activity leaks the data, the leak will be found.

These test cases, however, also include other, unrelated activities that never receive the intent. Since FlowDroid does not perform intent resolution, it does not connect the sender and receiver activities. Therefore, it finds a false positive in the unrelated activity that has code to leak the incoming data, bus is never triggered at runtime.

ComponentNotInManifest1
The actual component to which the intent containing the sensitive data is sent is not declared in the manifest file. FlowDroid correct discovers this and does not find a false positive in that activity. However, there is another unrelated activity which is declared in the manifest file, but which is not the receiver of the intent. As FlowDroid does not perform intent resolution, it reports the data flow inside this unrelated component from reading the incoming intent to leaking it as a leak.

EventOrdering1
This test case suffers from the same issues as `ActivityCommunication2-8` which explains the false positive. Additionally, a false negative arises, because FlowDroid does not handle taint across external resources. In the test case, sensitive data is written to Android's *Shared Preferences* infrastructure. When the app is restarted, the data is read back and leaked. One could avoid this false negative by defining the shared preference reading methods as sources which over-approximates the possible tainted data. Still, one would not be able to capture the precise flow *through* this external resource.

ServiceCommunication1
This test case starts a new service and passes an implementation of Android's `ServiceConnection` interface to it. Through this interface, the starting activity and the service can communicate. More specifically, this interface is used to exchange `IBinder` implementations for afterwards running transactions with the service. FlowDroid does not support such communication yet and cannot over-approximate it either.

SharedPreferences1
This test case writes sensitive data into the shared preferences in one activity and reads this data back in another activity where it is leaked. As in the `EventOrdering1` test case, FlowDroid does not support taint propagation over external resources.

Singletons1
This test case requires component interleaving. One activity clears a field in a singleton when started. When a second activity starts, it fills the field with tainted data. The data is leaked when the first activity is stopped. Since FlowDroid simlautes the component lifecycles one after another, this interleaving is not supported and thus, no leak is detected.

7.3.7 Lifecycle

ActivityEventSequence2
This test case was contributed by a different research group. We were unable to reproduce the data flow claimed in the test case's documentation during our tests with the app. We therefore do not count a false positive even though FlowDroid does not find the proclaimed leak.

BroadcastReceiverLifecycle2
In this test case, a false positive occurs, because FlowDroid models components sequentially without any dependencies in the dummy main method. For this test case to be analyzed correctly, the lifecycles of the main activity and the dynamically-registered broadcast receiver would, however, need to be interleaved.

FragmentLifecycle1 & 2
FlowDroid does not yet support Android's fragments. Aside from the four component types (activity, service, content provider, and broadcast receiver) plus the application class, this would be another distinct lifecycle to be implemented in the entry point creator. Fragments are complex since they are tightly integrated into the lifecycle of their hosting activity.

ServiceEventSequence1
The `service` component has two different lifecycles. Users can send commands to the service individually, or, alternatively, bind to the service once and then use this open connection to exchange data with the service. This test case combines both uses and mixes the two lifecycles which is not supported by FlowDroid.

7.3.8 General Java

Exceptions3
In this test case, the leak only happens if an `ArrayIndexOutOfBoundsException` is thrown. The code can, however, not throw an exception because the it only accesses an array at a constant index that is in the range of the array. FlowDroid over-approximates the set of exceptions that can be thrown at array accesses and thus assumes that the index might be illegal though it is not.

Serialization1
This test case requires a more complex library handling than FlowDroid's easy taint wrapper can offer. The easy taint wrapper can only taint the base object when a parameter of a method call is tainted. If the base object, however, references another object to which it passes on the tainted data; this cannot be modeled. To support such complex libraries, we propose StubDroid, see Section 6.

StaticInitialization1 & 3
FlowDroid depends on Soot to create the callgraph for the app based on the dummy main method that FlowDroid's entry point creator generates. In Soot, static code blocks are, however, handled imprecisely. This can lead to the static block being executed at the wrong time, hiding the leak in these two test cases.

StringFormatter1
Same reason as `Serialization1`, gets solved with StubDroid.

VirtualDispatch3
In this test case, a variable that is declared with an interface type is initialized through a factory method. Afterwards, data is retrieved from the object using a getter method and this data is leaked. Depending on the concrete implementation of the interface type chosen by the factory method, a leaks happens or not. Though the factory method only instantiates the type that does not return any sensitive information, the SPARK callgraph algorithm built into Soot delivers imprecise information on the possible types for the result of the factory method. Therefore, it constructs outgoing call edges to all implementations of the interface, including those for which a leak occurs.

7.3.9 Miscellaneous Android-Specific

PrivateDataLeak3
The sensitive data is written into a file and read back. FlowDroid does not support such taint paths across external resources. One could declare file inputs as a source to over-approximate the file handling. In this case, the missed leak would still occur in our configuration, though, because one would need complex library handling. The over-approximation in combination with StubDroid would work.

7.3.10 Native Code

FlowDroid does not support tracking data flows through native code. Therefore, no leaks can be found for these benchmarks as explained in Section 4.10.

JavaIDFunction, SinkInNativeCode
This test case leaks sensitive data using JNI. The data is passed into a native method which again uses JNI to pass the data back to the sink in the (Dalvik-based) Android SDK. This sink call can thus not be detected by FlowDroid as the call site is not in the Dalvik part of the app.

NativeIDFunction
This test cases passes sensitive data through an identity function in native code which is not supported by FlowDroid. The taint propagation ends once the data arrives at the native call and thus never reaches the sink.

SinkInNativeLibCode
This test case is similar to `SinkInNativeCode`, but the call is not a method in the Dalvik-based part of Android SDK and called through JNI from native to Dalvik. Instead, native API methods are used to directly leak the data. Such native-level sinks cannot be detected by FlowDroid.

SourceInNativeCode
In this test case, the source is called from native code. The native library uses JNI to invoke a method in the Dalvik-based part of the Android SDK. The call site in the native code is not detectable by FlowDroid.

7.3.11 Reflection

FlowDroid inherits its support for reflective method calls from the underlying Soot framework. This support is limited to cases in which the types and names of classes referenced through reflection are known. For methods, the name and the parameter types must be available.

Reflection3
In this test case, the names of the class and method called through reflection are obtained from Android resource files and are not directly available as string constants. Therefore, Soot's built-in mechanism for resolving reflective method calls fails. The problem can be circumvented by first running the app through the Harvester tool [112], which resolves the reflective method calls and converts them into direct method invocations. The resulting app can then be processed by FlowDroid.

7.3.12 Reflection_ICC

ActivityCommunication2
This test case inherits the same issues as the original test case with the same name in the category for inter-component communication.

AllReflection, OnlyIntentReceive
These test cases use factory methods to obtain the base objects on which methods are afterward invoked using reflection. In that case, the SPARK callgraph algorithm cannot obtain the base type, and consequently, cannot find suitable candidate methods for the reflective call. Since the callees are library methods, no actual implementation is expected, but also for the taint wrappers to work, at least the correct signature of the receiver method must be known.

SharedPreferences1
This test case fails for the same reason as its counterpart from the `Inter-Component Connction` category that does not use reflection. FlowDroid does not support tracking taints across files, including the shared preferences XML file.

7.3.13 Self-Modification

FlowDroid only analyzes the Dalvik code inside the APK file. It does not parse or analyze any native code and thus cannot find out whether the native code changes the behavior of the Dalvik code inside the phone's memory. Therefore, FlowDroid must assume that the Dalvik VM (or the ART runtime) always executes the original code from the APK's `classes.dex` file.

7.3.14 Unreachable Code

FlowDroid performs an interprocedural constant propagation and folding as explained in Section 4.14. While this technique is able to identify code that is unreachable due to conditionals, e.g., on the outcome of simple arithmetic computations, it does contain a model of Java's mathematical library functions. The test cases in this category compute a random integer with a specific maximum value and then check whether the outcome is larger than a value higher than this upper limit. While this condition can never become true, FlowDroid lacks a model for the relationship between the maximum value passed to the randomization function and the return value of that function. Therefore, the analysis conservatively assumes both branches of the conditional to be feasible and a false positive occurs.

7.4 Discussion

In the original paper on FlowDroid [9], we also compared the tool's precision and recall to the results we achieved with HP Fortify, another commercial static analysis tool. Unfortunately, Fortify was no longer available to us when conducting the experiments for this thesis. On the original DroidBench version 1.0 published together with the FlowDroid paper, Fortify achieved a precision of 81% and a recall of 61%. These results can only be assessed in comparison with the old data for IBM AppScan Source and FlowDroid, though. The structure of DroidBench has shifted by adding new test cases that make it significantly harder to achieve high precision and recall in version 3.0 of the micro-benchmark suite than in version 1.0. On DroidBench 1.0, FlowDroid achieved a precision of 86% and a recall of 93%. In our old experiments, we used version 8.3 of IBM AppScan Source which achieved a precision of 74% and a recall of 50%.

In the current experiments, we can observe that FlowDroid still outperforms IBM AppScan Source with regard to both precision and recall, regardless of whether scan coverage findings are counted in or not. Interestingly, including scan coverage findings into the AppScan Source results increases not only the recall of the tool, but also the precision, because with the massively higher number of correct findings, the only slightly increased number of false positives is no longer as significant.

We have found that DroidSafe was unable to complete the analysis for 19 of the 190 DroidBench test cases, either because of timeouts or because of uncaught exceptions. In our numbers for precision and recall, we exclude those test cases. The precision of FlowDroid (86.81%) is lower than the one of DroidSafe (90.67%), but FlowDroid has a higher recall (84.04%) than DroidSafe (72.43%), i.e., detects more leaks. In the F-measure that aggregates precision and recall, FlowDroid achieves a slightly higher result (0.86) than DroidSafe (0.80). The numbers are based on the total number of leaks to be found. If a tool does not complete the analysis of a test case, it is counted as if it had found nothing. This approach biases the metrics towards higher precision (because it assumes no false positives in that case), but lower recall (because it assumes all true leaks to be missed). If we compute the metrics only on those apps for which the respective tool completed the analysis and exclude the failing test cases from the DroidBench test suite when evaluating the corresponding tool, the recall of DroidSafe rises to 80% and the F-measure to 0.85. Both values are closer to, but still lower than the results achieved by FlowDroid. Furthermore, we can draw the conclusion that FlowDroid is the more stable and more mature analysis tool. We also point out that DroidSafe was published in 2015, when FlowDroid was already approximately available for one year, allwong the DroidSafe project to build upon our work.

Concerning recall, we notice that DroidSafe provides a model for fragments, which accounts for four leaks FlowDroid misses in the *Lifecycle* category of DroidBench. Providing such a model for FlowDroid is the subject of an ongoing research project. On the other hand, DroidSafe does not seem to support implicit flows, which leads to seven missed flows in the *Implicit Flow* category of DroidBench. Concerning precision, we find that FlowDroid yields nine false positives in the *Inter-Component Communication* category of DroidBench, because it analyzes components independently and thus cannot detect when a component inside the app never actually receives data. We point out that two publications, namely ICCTA [86] and DidFail [77] address this issue by extending FlowDroid with full support for inter-component and inter-app data flow tracking. We re-evaluated the apps in the *Inter-Component Communication* category with ICCTA. Without it, the FlowDroid analysis yields 15 true leaks (⊛), 4 false negatives (○), and 9 false positives (⋆) in this category. With ICCTA, the results change to 14 true leaks (⊛), 5 false negatives (○), and 1 false positive (⋆). The ICC models, which define the inter-component callgraph edges, were created with IC3 [103]. On this test suite, ICCTA improves the precision (from 86.81% to 90.75% in total), but decreases the recall (from 84.04% to 83.51% in total). The F-Measure, which is a combined measure for precision and recall, remains almost unchanged; it only increases slightly from 0.86 to 0.87. With more precise and complete ICC models, the accuracy of the analysis can likely be improved even further. The false negatives from which FlowDroid and ICCTA suffer for the *ActivityCommunication2* test case, for example, stem from a missing ICC edge in the model.

JoDroid, which is based on system dependence graphs instead of taint tracking, shows lower precision and recall than FlowDroid. This is surprising as Joana, the tool on which JoDroid is built, was constructed with soundness in mind. Nevertheless, we find that even for such a tool, the use of reflection, native code, or API methods not modeled in the tool can lead to cases of under-approximation. Out of 190 DroidBench test cases in total, 12 apps could not be analyzed, mostly due to unhandled exceptions in the tool. In two of these cases, we, however, also found that the tool did not terminate in realistic time. We also had to increase the maximum heap space allotted to the SDG generation from its 2 GB default, which proved to be insufficient for some of the DroidBench test cases. We used a new maximum heap space of 25 GB, but did not further investigate whether a lower amount would have been sufficient as well. For the information flow analysis on the generated SDG, we found the Joana GUI to actually

use around 9 GB of memory, with some `.pdg` files[36] growing up to over 150 MB for micro-benchmarks apps of less than 1 MB and with only a handful of lines of user code. We conclude that FlowDroid not only appears to be more precise and complete, but also more stable and scalable.

[36] The tool seems to sometimes refer to PDGs (program dependence graphs) and sometimes to SDGs (system dependence graphs).

Setting	Value	Default	Explanation	Notes
Access Path Length	1	No (Default: 5)	See Section 3.3	Scalability on large apps
Symbolic Access Paths	Enabled	Yes	Section 3.3	
Path Agnostic Results	Enabled	Yes	Section 4.13	
Data Type Propagation	Enabled	Yes	Section 4.11	
Implicit Flow Tracking	Disabled	Yes	Section 4.6	
Track Static Fields	Enabled	Yes	Section 4.2	
Track Exceptional Flows	Enabled	Yes	Section 4.5	
Array Size Tainting	Disabled	Yes	Section 4.4	
Alias Algorithm	Flow-Sensitive	Yes	Section 4.8	
Flow Sensitive Aliasing	Disabled[38]	No	Section 4.8.4	Scalability on large apps
Callgraph Algorithm	SPARK	Yes	Section 4.2.6	
Code Optimization	Constant Prop.	Yes	Section 4.14	
Android Callbacks	Enabled	Yes	Section 5.3	
Callback Sources	Enabled	Yes	Section 5.3.7	
Callback Analyzer	Iterative	Yes	Section 5.3.4	
Layout Matching	Sensitive Only	Yes	Section 5.2	

Table 7: Default Configuration for FlowDroid Performance Evaluation

8 FlowDroid Performance Evaluation

In this section, we evaluate the performance of FlowDroid on real-world Android apps. Note that we do not evaluate the precision or recall of the analysis, because this is hard to accomplish on real-world apps for which no ground truth exists and which are significantly too large for manual reverse-engineering. For measurements on FlowDroid's precision and recall on the DroidBench micro-benchmark suite, please refer to Section 7.2. For the performance evaluation, we chose a set of popular apps from the Google Play Store that are installed on many devices and for which a privacy analysis concerning data flows is thus of interest to many users. These popular apps also have a rather large code size and use many different language constructs, which again makes them interesting targets for assessing a static analysis tool. For sampling the apps, we used the list of most downloaded apps from Wikipedia[37] and sampled it for apps from different categories including games, social media, productivity, and communication. Additionally, we added further popular apps from other sources.

8.1 Default Configuration

Recall that the performance of the data flow analysis depends various factors, both in the platform-agnostic part of the tracker as well as in the Android-specific implementations. By default, the test program shipped with FlowDroid already configures the solver with an option set that makes a reasonable tradeoff between precision, recall, and performance. Additionally, we specifiy some additional settings in order to be able to cope with even very large apps. Our complete configuration is shown in Table 7. For evaluating the effect of specific features or design decisions of the data flow tracker, we adapt individual settings in the following sections, but always start from the default we present here. The evaluation of StubDroid in Section 6.7 has already shown that the performance of the data flow analysis is not significantly different when using the precise StubDroid summaries as opposed to FlowDroid's default *Easy Taint Wrapper*. We therefore chose to take this default library handling (i.e., the Easy Taint Wrapper) for the performance tests for the sake of simplicity. This avoids the the requirement to create StubDroid summaries for each library used in any of the apps we take for the performance analysis for all the different Android versions against which they were built. We would like to point out, though, that fulfilling these requirements would be a one-time task that is computationally feasible in realistic time in a productive setting.

[37] Source: `https://en.wikipedia.org/wiki/List_of_most_downloaded_Android_applications`

[38] Note that we still use FlowDroid's flow-sensitive alias analysis, we only disable setting activation statements. The IFDS backwards analysis is still in use, it only runs the risk of becoming flow-insensitive when handing over taints between forward and backward solver.

For sources and sinks, we used the default file shipped with FlowDroid. This file only contains a subset of the total sources and sinks detected by SuSi for practicability reasons. The more sources there are (which are also used inside the app under analysis), the more data needs to be tracked. Specifying hundreds of sources can thus make the analysis infeasible on such large apps. The default subset of sources and sinks includes the most common ones that are commonly tracked by academic or industrial data flow analyzers. We refer the reader to Table 8 for the complete list of sources and to Table 9 for the complete list of sinks. For the sake of brevity, we only give the names of the methods and not the complete parameter lists. If a method has multiple overloads, all of them are considered to be sources or sinks, respectively. Further note that, since FlowDroid does not support inter-component communication by default, we over-approximate the respective sources and sinks. All data that is received through an intent is considered as tainted and all data that leaves the current component through an intent is considered as leaked.

We conducted all test runs on a server computed with 64 Intel Xeon E5-4650 CPUs running at 2.7 GHz base frequency[39]. The machine is equipped with 1 TB of physical main memory, but we limit the maximum heap size of the Java process running the data flow analysis to 25 GB by default. For some apps, this limit of 25 GB was not sufficient, though. In those cases we used a larger maximum heap size which we denote when we report the measurement data for the respective app. When an app analysis could not be completed, we first increased the maximum heap size to 50 GB and then, if the analysis still did not complete, to 100 GB. Note that even in cases in which the user doesn't have enough memory to complete the full analysis, he can use the memory thresholding approaches explained in Section 4.12.3 to obtain those results that are available before the analysis runs out of memory. The same applies to long-running analyses. The user can specify a time budget and obtain those results that have been computed before the time budget is used up. In many cases, even a subset of results already gives the user certain hints as to whether the app is careful not to leak any privacy-sensitive information or whether it synchronizes everything with the cloud. For our experiments, we relied on memory thresholding (without restricting the time budget) when even the increased heap size of 100 GB was not sufficient. In those cases, we allowed FlowDroid to abort the analysis earily and report partial results. When presenting evaluation data, we mark such cases in addition to the increased heap size. Note that we cannot give all analyses a maximum heap size of 100 GB right away, because this would lead to increased memory usage even for apps which don't require so much memory. This effect happens, because the JVM rather uses the available memory, instead of triggering garbage collection cycles. Therefore, only a suitable limit gives proper evidence on the real memory consumption. Furthermore, the amount of available memory also influences the runtime, because frequent garbage collector cycles require time in addition to the normal taint propagation. Therefore, applying a sufficient, but not oversized maximum heap size, is a reasonable compromise for obtaining realistic time and memory measurements.

All tests were executed on the Oracle JDK version 1.8-05. To account for non-deterministic effects introduced through thread scheduling or the JVM's just-in-time compiler, we ran each test ten times and averaged over the measurements. Since the computation server is a shared resource, external effects such as the workloads imposed on the server by other users can negatively affect the FlowDroid runtime. Since FlowDroid uses all cores available on the machine, even single-threaded external workloads can have extend the time FlowDroid requires for the analysis. To account for such effects, we removed obvious outliers (e.g., single runs that took ten times longer than the other runs with the same configuration on the same app), from the averaged timings we report.

8.2 Basic Performance Evaluation

Table 10 shows the apps that we have used for assessing the performance and scalability of FlowDroid. For each app, we specify the common name as displayed by the app itself or the app store from which it was downloaded, the package name that uniquely identifies the app, and the number of Jimple statements contained in the app as an approximate measure of the size of the app. Note that the APK file size alone is not a good measure for the code size, because the APK file can also contain, depending on the app, large images, audio files, especially for games or virtual reality apps. For counting the number of Jimple statements, we summed up the number of units in all methods contained directly in the app's `classes.dex` file. This is an approximation, because it does not take into account which methods are actually reachable. Especially in the case of libraries or SDKs compiled into the app, a number of methods are expected to be unused, because the app is unlikely to use each and every method of the library. Still, this counting is a good approximation for the app size. For ruling out unused methods, one would need to create a dummy main method based on the app's lifecycle model, create a callgraph from it, and then calculate

[39] The CPU uses adaptive clock speeds depending on system load and CPU core temperature. We can therefore only specify its base clocking.

Category	Class	Methods
Location Data	android.location.Location	getLatitude(), getLongitude()
	android.location.LocationManager	getLastKnownLocation()
	android.telephony.gsm.GsmCellLocation	getCid(), getLac()
	java.util.Locale	getCountry()
	java.util.Calendar	getTimeZone()
Identification Data	android.telephony.TelephonyManager	getDeviceId(), getSubscriberId(), getSimSerialNumber(), getLine1Number()
	android.bluetooth.BluetoothAdapter	getAddress()
	android.net.wifi.WifiInfo	getMacAddress(), getSSID()
Browsing	android.provider.Browser	getAllBookmarks(), getAllVisitedUrls()
Network Communication	java.net.URLConnection	getInputStream()
	org.apache.http.HttpResponse	getEntity()
	java.net.URL	openConnection()
Inter-Component Communication	android.app.PendingIntent	getActivity(), getBroadcast(), getService()
	android.os.Handler	obtainMessage()
Phone Configuration	android.content.pm.PackageManager	getInstalledApplications(), getInstalledPackages() queryIntentActivities(), queryIntentServices(), queryBroadcastReceivers(), queryContentProviders()
Shared Preferences	android.content.SharedPreferences	getDefaultSharedPreferences()
Databases	android.database.sqlite.SQLiteDatabase	query()

Table 8: Sources for FlowDroid Performance Evaluation

Category	Class	Methods
Log Output	android.util.Log	d(), e(), i(), v(), w(), wtf()
Inter-Component Communication	android.os.Bundle	put*()
	android.content.Intent	setAction(), setClassName(), setComponent()
	android.content.Context	sendBroadcast(), sendOrderedBroadcast(), startActivity(), startActivities(), startService(), bindService()
	android.os.Handler	sendMessage()
IO Streams	java.io.OutputStream	write()
	java.io.OutputStreamWriter	append()
	java.io.Writer	write(), append()
Network Communication	java.net.URL	openConnection()
	java.net.URLConnection	setRequestProperty()
	java.net.Socket	connect()
SMS Communication	android.telephony.SmsManager	sendTextMessage(), sendDataMessage(), sendMultipartTextMessage()
Shared Preferences	android.content.SharedPreferences$Editor	put*()
Processes	java.lang.ProcessBuilder	start()

Table 9: Sinks for FlowDroid Performance Evaluation

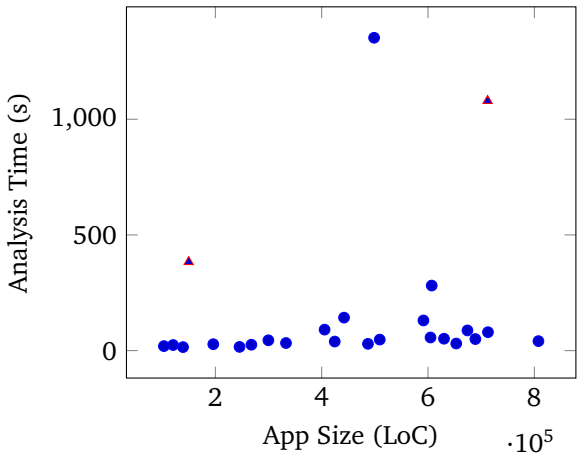
App	Package Name	Code Size	Sources	Downloads
Adobe Reader	com.adobe.reader	139,236	131	100M - 500M
Amazon Kindle	com.amazon.kindle	606,941	224	100M - 500M
Angry Birds	com.rovio.angrybirds	487,050	94	100M - 500M
AutoScout24	com.autoscout24	712,769	98	10M - 50M
BurgerKing	de.burgerking.kingfinder	653,064	52	5M - 10M
CNN	com.cnn.mobile.android.phone	299,743	330	10M - 50M
eBay	com.ebay.mobile	712,068	689	100M - 500M
Facebook	com.facebook.katana	442,066	578	1B - 5B
Facebook Messenger	com.facebook.orca	120,594	55	1B - 5B
Google Plus	com.google.android.apps.plus	807,668	1	1B - 5B
Instagram	com.instagram.android	498,576	88	1B - 5B
LinkedIn	com.linkedin.android	149,888	480	50M - 100M
Microsoft Outlook	com.microsoft.office.outlook	630,038	208	50M - 100M
Microsoft Word	com.microsoft.office.word	689,069	69	100M - 500M
Netflix	com.netflix.mediaclient	405,474	159	100M - 500M
Offi Journey Planner	de.schildbach.oeffi	103,265	93	5M - 10M
Opera	com.opera.browser	332,884	123	100M - 500M
PayPal	com.paypal.android.p2pmobile	674,168	188	10M - 50M
Pinterest	com.pinterest	591,387	178	100M - 500M
Pokemon Go	com.nianticlabs.pokemongo	424,550	72	100M - 500M
Skype	com.skype.raider	196,123	193	500M - 1B
Telekom Mail	de.telekom.mail	267,807	144	1M - 5M
Tinder	com.tinder	509,192	173	50M - 100M
VLC for Android	org.videolan.vlc	245,513	96	50M - 100M
Wetter.com	com.wetter.androidclient	604,701	221	10M - 50M

Table 10: Real-Wold Apps for FlowDroid Performance Evaluation

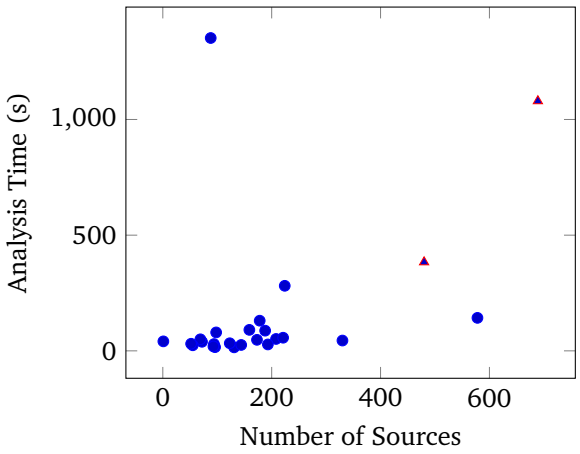
App	Heap Limit (GB)	Runtime (s)	Memory (MB)		Completed	Leaks
			Min.	Avg.		
Adobe Reader	25	15.15	520.99	1,765.29	Yes	39
Amazon Kindle	25	281.15	7,906.42	12,994.42	Yes	99
Angry Birds	25	29.24	1,005.21	3,642.33	Yes	39
AutoScout24	25	79.66	1,067.22	4,839.29	Yes	43
BurgerKing	25	30.64	1,007.49	4,157.71	Yes	32
CCN	25	44.55	1,274.36	3,197.52	Yes	184
eBay	100	1,079.66	72,695.56	78,507.27	No (Memory)	269
Facebook	25	142.54	2,585.28	3,143.53	Yes	154
Facebook Messenger	25	24.08	522.36	1,997.73	Yes	14
Google Plus	25	40.99	1,182.53	5,033.85	Yes	1
Instagram	100	1,352.12	47,320.77	49,322.62	Yes	132
LinkedIn	100	383.38	66,957.34	74,199.98	No (Memory)	131
Microsoft Outlook	25	51.41	1,535.30	4,969.30	Yes	57
Microsoft Word	25	49.89	2,029.67	7,656.06	Yes	26
Netflix	25	90.71	2,034.74	7,536.65	Yes	74
Offi Journey Planner	25	19.39	637.28	2,044.04	Yes	13
Opera	25	32.83	1,116.89	2,935.80	Yes	57
PayPal	25	87.24	1,645.72	5,461.64	Yes	90
Pinterest	25	130.35	2,179.57	5,383.73	Yes	107
Pokemon GO	25	39.17	1,418.35	5422.42	Yes	20
Skype	25	27.67	552.71	808.71	Yes	21
Telekom Mail	25	25.51	734.18	2,742.11	Yes	66
Tinder	25	47.63	956.15	1,170.53	Yes	61
VLC	25	16.09	630.61	2142.38	Yes	15
Wetter.com	25	56.75	1,644.61	4,921.25	Yes	85

Table 11: FlowDroid Performance With Default Configuration

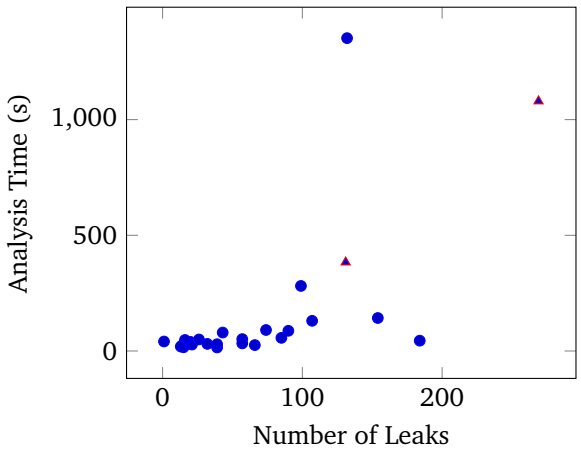
reachability. This lifecycle model, however, already depends on the configuration of FlowDroid and the resulting unit count and is thus not a good baseline. Further, note that the unit count only takes the units directly inside `classes.dex` into account, but no inside any additional `dex` files that might be contained in the app. This in in line with FlowDroid's behavior (the data flow tracker cannot deal with additional `dex` files either), and thus does not affect the precision of the app size approximation for the task at hand. For each app, we also list how many calls to methods we consider as sources they contain. This is relevant when we discuss the performance and scalability of the data flow analysis later in this section. To further motivate the relevance of the apps we have chosen, we furthermore report the number of times the app was downloaded from the official Google Play Store. This is not an exact number, but a range, because the Play Store only provides such ranges and not precise counts to the public.
In the basic performance evaluation conducted in this section, we use the default settings explained in Section 8.1 without any modifications. These numbers are later used as a baseline when assessing the impact of the various features and configuration settings on the performance of the data flow analysis. Table 11 shows the average time and memory consumption of the analysis. Times are given in seconds, memory sizes in megabytes except for the maximum heap size which is given in gigabytes. Note that the average memory consumption is not necessarily the amount of memory necessary to conduct the analysis. With tighter memory constraints, the JVM might perform more aggressive garbage collection and decrease the its memory usage, potentially (but not necessarily) affecting performance. In general, we find that the JVM rather uses more of the available heap size than conducting a garbage collector cycle as long as space is available. We further noticed that the overall usage of the heap space allotted to the JVM increased over the 10 measurements, even though we manually triggered a garbage collector run through a call to `System.gc()` after each run. Therefore, we also report the minimum memory consumption of the ten runs we conducted per app in Table 11.
From tables 10 and 11, it also becomes evident that the number of Jimple statement inside the app is not proportional to the computation time or memory consumption of the analysis. While the Tinder app has more lines of code than the Facebook app, it only takes about 33% of the computation time and requires only about 37% of the



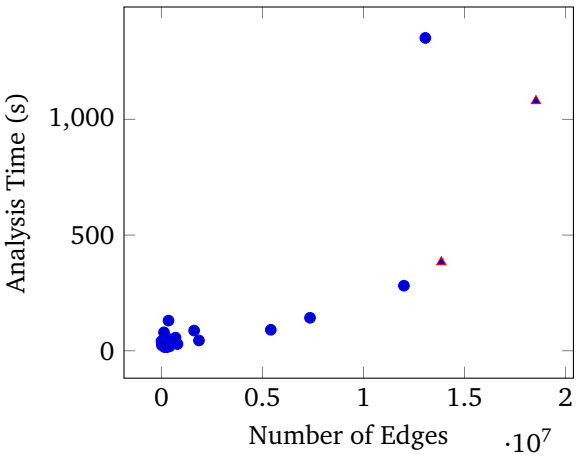
(a) Code Size vs. Analysis Time



(b) Source Count vs. Analysis Time



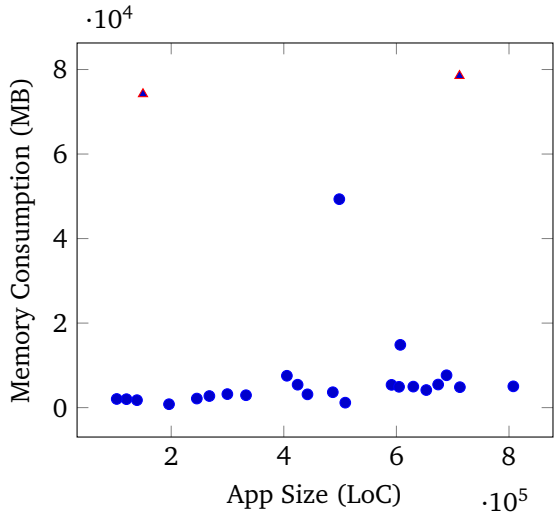
(c) Leak Count vs. Analysis Time



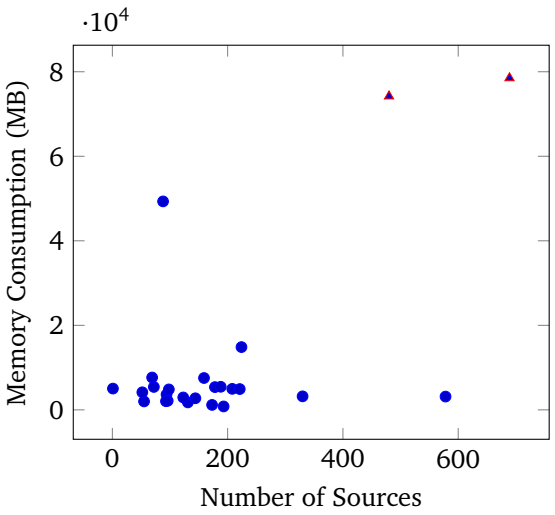
(d) Edge Count vs. Analysis Time

Figure 20: Factors That Influence Analysis Time

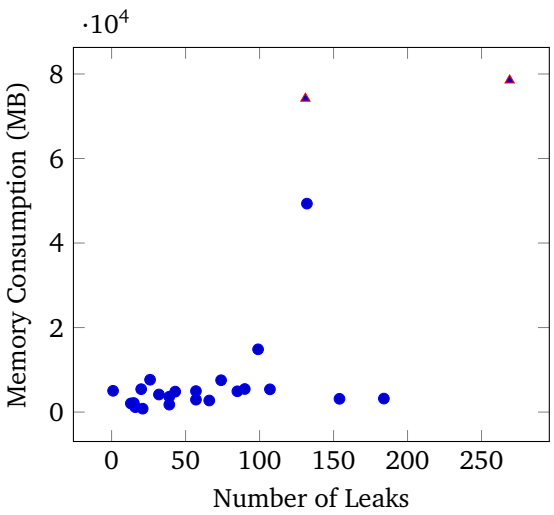
memory. Figure 20(a) shows a plot of the analysis time (y axis) over the app's code size (x axis). The apps that finished without a timeout or memory exhaustion are plotted with blue circles. The apps for which the analysis was aborted before it was fully completed are shown with read triangles. From the plot, we can only conclude that long computation times usually only arise above a certain code size. Even beyond this size, there are, however, still apps for which the analysis terminates quickly. Another candidate for aproximating the analysis time before actually conducting the analysis is the number of sources. As shown in Figure 20(b), the time required for the analysis, is, however, not directly correlated with the number of sources either. For most apps, even a higher number of sources does not significantly increase the computation time in practice. Some apps with a rather low number of sources, however, require a very high computation time. In total, given an arbitrary app, there is no easy way for an analyst to approximate how long the taint analysis will take.
In Figure 20(c), we plot the the analysis time against the number of leaks finally detected by the static data flow analysis. Again, it becomes visible that there is no direct correlation.There are apps with a relatively small number of leaks such as Youtube with only 23 leaks that nevertheless take a long time to analyze as shown by the upper left blue dot in the figure. Other apps, with more than twice the number of leaks, on the other hand, can be analyzed in a small fraction of the time.
Besides the time required to conduct the analysis, the memory consumption of the analysis is also important to assess the feasibility of the FlowDroid approach. Table 11 shows FlowDroid ran out of memory for some of the apps. As explained in Section 8.1, we increased the maximum heap size in two steps for such cases. During our experiments, we observed that if the analysis ran out of memory for an app with a maximum heap size of 25 GB, it usually also ran out of memory even with an increased maximum heap size of 100 GB. This effect is likely due to over-tainting. If the analysis taints a significant portion of the app for some reason, this happens similarly at many positions inside the app, leading to a large number of spurious taint abstractions. While theoretically bounded, this over-tainting will in practice prevent the taint tracker from completing the analysis given any realistic amount of memory and time. In such cases, we relied on FlowDroid's memory thresholding technique to still receive partial results. As we explained in Section 4.12.3, these partial results are in many cases still sufficient to give the analyst a hint concerning the general privacy-friendliness of the app.
For an analyst, it would be advantageous to know upfront whether an app can be analyzed in full or whether such an over-tainting case will occur. Furthermore, for those apps on which the analysis can be completed fully, it would be beneficial to have an a-priori approximation of the amount of memory that will be required. However, similar to the time required for the analysis, there is no simple correlation that could be exploited. In the case of the Instagram app, for instance, we had to increase the maximum heap size of the JVM (now using a limit of 100 GB) for the analysis to succeed. As shown in Table 11, the actual amount of memory used by the analysis is also above the normally-allotted limit of 25 GB. Notably, the 100 GB heap size was sufficient for completing the analysis in this case. This large amount of memory was required, although the code size is comparable to other apps such as Facebook or Microsoft Outlook, for which the analysis terminated quickly within the original heap limit of 25 GB.
Figure 21(a) shows the memory consumption (y axis) over the number of Jimple statements in the respective app (x axis). There is no clear correlation between the two factors. Some large apps can be analyzed with only little memory, while some smaller ones even lead to memory exhaustion when increasing the memory limit to 100 GB. Note that for those apps in Table 11 that were aborted because of memory exhaustion, the maximum amount of memory consumed is still below the maximum heap size. This happens, because FlowDroid's memory thresholding must abort the analysis early and retain some of the memory, such that it is still able to shutdown the IFDS solver and process the results, before the JVM is terminated with an `OutOfMemoryError` as explained in Section 4.12.3.
Figure 21(b) shows the memory consumption (y axis) over the number of source call sites detected in the app (x axis). From the results of our experiments, one can only conclude that increasing the number of sources beyond a certain threshold increases the likelihood of memory exhaustion. Still, some apps with a relatively low number of sources have a high memory consumption and some with a large number of apps require only little memory for analysis. Similarly to the analysis time, there is no easy formula for, given an arbitrary app, approximating the memory consumption of the analysis beforehand. When taking the result of the analysis into account and comparing the number of leaks detected by the data flow tracker with the memory consumption as shown in Figure 21(c), the result is similarly unconclusive. More leaks beyond a certain threshold increase the likelihood of the analysis running out of memory, but there is no general correlation between the two values.
For both analysis time and memory consumption, we conclude that the structure of the app, i.e., the number of statements over which tainted data needs to actually be propoagated (rather than statements present in the app in general), has a much greater impact than code size, number of sources, and even number of leaks. Even if an app is very large, as long as taints only need be propagated through comparingly small fractions of the code, the



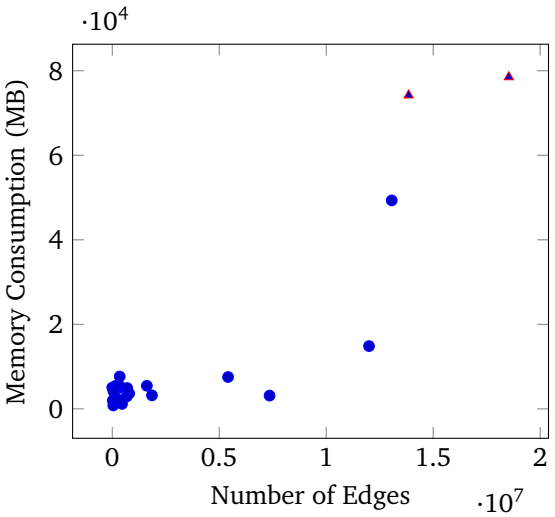
(a) Code Size vs. Memory Consumption



(b) Source Count vs. Memory Consumption



(c) Leak Count vs. Memory Consumption



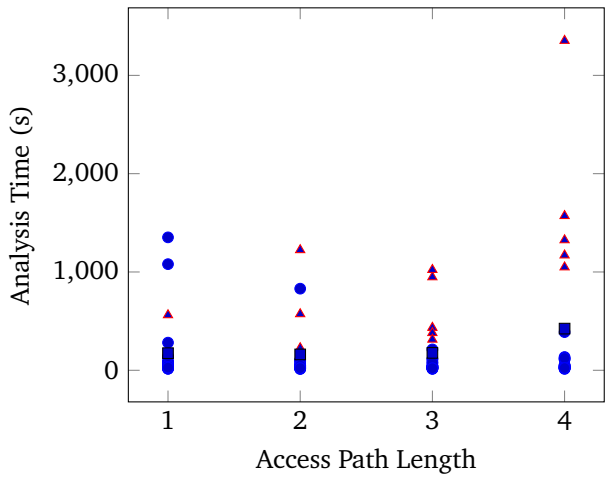
(d) Edge Count vs. Memory Consumption

Figure 21: Factors That Influence Analysis Memory Consumption

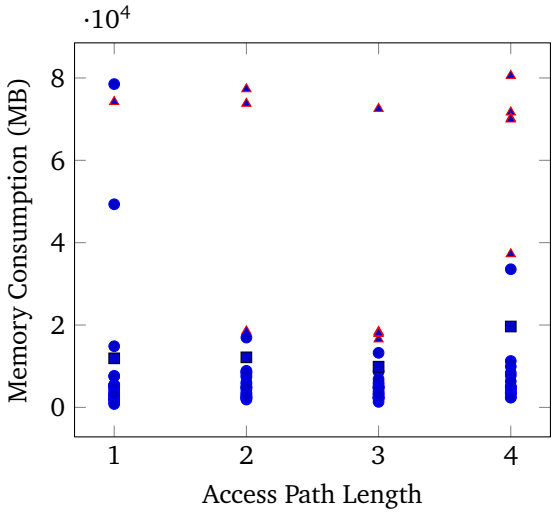
analysis can still be completed with low time and memory consumption. If the taints are propagated over large code bases, this leads to scalability issues. In terms of the IFDS algorithm, we count the number of edges in the exploded supergraph. Each edge corresponds to one evaluation of a flow function. Such an evaluation happens whenever a new combination of context and statement appears in the solver's worklist, i.e., tainted data arrives at a statement that hasn't previously been seen under the current context. Obviously, more flow function evaluations increase the analysis runtime. Figure 20(d) shows a plot of analysis time (y axis) over the number of IFDS edges (x axis) which very well visualizes this dependency. Furthermore, each generated triple of context, statement, and result of the flow function must be stored to be able to detect when a fixed point has been reached. Each stored triple increases the memory consumption as shown in Figure 21(d). The dependency between the number of edges and the memory consumption follows a similar shape as the dependency between the number of edges and the time required for the analysis. We furthermore observe that all apps for which the analysis was aborted early due to memory exhaustion are on the upper right side of the plot, i.e., have the largest edge count. The plot suggests an exponential grow in memory consumption over the number of propagated egdes. Beyond a certain threshold, not enough memory is available anymore. In the plot, we can also see that FlowDroid's memory thresholding technique stops the analysis once it reaches about 80% of the total available memory.
Also, the size and complexity of the Android lifecycle is an important factor, because it directly influences the number of edges to be propagated. If the app's code is distributed over more components or callbacks there are more possible paths through the code, because of the possible interleavings of the respective callback methods. This can lead to the same code being reachable through a larger number of distinct contexts. If the code on the other hand, is straight-line code inside a single callback, there is only one possible execution sequence and context. From a technical point of view, the connection between the number of edges on one hand and the time and memory consumption on the other hand is understandable given how the IFDS solver works. On the downside, this makes it hard for the analyst to have an a-priori approximation of the effort he needs to spend on a given app in terms of time and memory. The number of edges over which taint abstractions are propagated is usually only available after the analysis has already been conducted, rendering it useless for a-priori predictions.

8.3 Adapting The Access Path Length

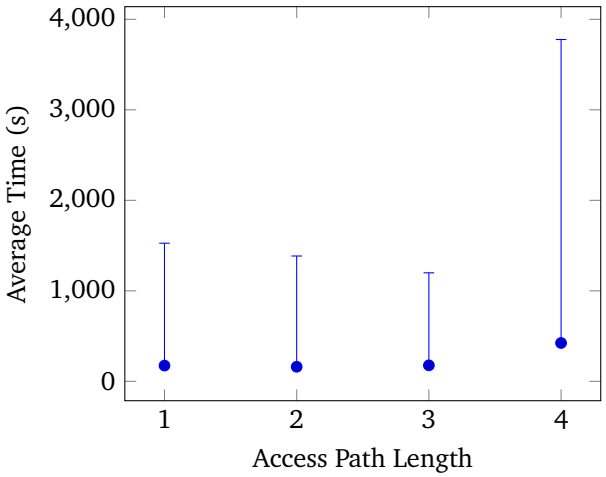
In this section, we take the base configuration from Section 8.2 and measure the effect of gradually increasing the access path length on performance and memory consumption. We re-use the same maximum heap size that was sufficient for the base case reported in Table 11. We started our analyses with the maximum heap size with which the base evaluation was successfully terminated. If, due to the increase in access path length, the analysis of an app then failed due to memory exhaustion, we iteratively increased the maximum heap size in the say way as we did for the base evaluation. If an app fails with 25 GB, it is re-tried with 50 GB, and if this still fails, with 100 GB. If even a maximum heap size of 100 GB is not sufficient for completing the analysis, we report it as a case of memory exhaustion and use the runtime and memory consumption actually consumed until the analysis was interrupted by FlowDroid's memory thresholding technique. We also limit the analysis to 30 minutes for each of the 10 test runs. We note that apps, for which the analysis does not complete within 30 minutes, usually cannot be analyzed even when given several hours.
When increasing the access path length, we expect to see fewer reported leaks, because false positives that result from over-tainting are less likely to occur. If the actual tainted access path would have been `a.b.c`, the analysis can only precisely pinpoint the respective memory location if it runs with an access path length greater than or equal to two. If access paths are instead truncated at length one, all memory locations reachable through `a.b.*` would be tainted, including a potential false taint such as `a.b.d`. While this general rule of the number of leaks being inversely correlated with the access path length holds true for most apps in our experimental app, we also noticed cases in which longer access path lead to *more* results. In the *Wetter.com* app, FlowDroid, for instance, discovered 85 leaks when run with an access path length of one. After increasing the length to two, it discovered 101 leaks. Such effects are related to the over-taint filtering built into FlowDroid (see Section 5.4.1). With this technique, the memory manager detects if whole Android components are tainted, which is usually a sign of serious over-tainting that has the chance to make the complete taint analysis infeasible in any realistic amount of time and memory. Therefore, such taints are killed. In the example above, assume that `a.b` refers to an Android activity. If the field `a.b.c` is meant to be tainted, but the access path is shortened to `a.b.*`, this will taint the whole activity and be deleted by the overtaint filtering. On the downside, this also means that the originally-intended memory object `a.b.c` is no longer tainted either. In such a case, increasing the maximum access path length decreases the chances of taints being killed by the filter and can lead to more leaks being detected.



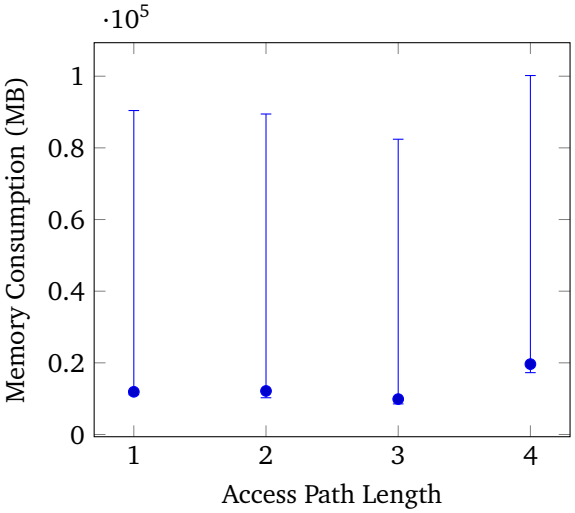
(a) Access Path Length vs. Analysis Time



(b) Access Path Length vs. Memory Consumption



(c) Access Path Length vs. Analysis Time (2)



(d) Access Path Length vs. Memory Consumption (2)

Figure 22: The Effect of the Access Path Length

From the theory, it is expected that the time and memory consumption grow together with the access path length. On the other hand, the increased precision of longer access paths can also reduce over-tainting and thus the number of taint abstractions that must be tracked. We find that, in total, the average increase in computation time is 32% from access path length one to two, 40% from length two to three, and 56% from length three to four. Memory increases by 69% from length one to two, by about 5% from length two to three, and by 75% from length three to four. For the *Instagram* app, we can only report a data point for access path length one, because the analysis did not finish in reasonable time for larger access paths. We aborted it after several hours. For some other apps, the analysis did not terminate beyond a certain access path length either, most commonly due to memory exhaustion. In such cases, the measured time and memory values do not represent the actual requirements of fully analyzing the respective apps. We therefore compute new averages for which we remove apps from the sample set once a memory exhaustion happens. With this reduced sample set, we arrive at an average increase in memory consumption by about 72% from access path length one to two, an increase by around 1% from length two to three, and one by about 40% from length three to four. With this adapted averaging metric, the time consumption increases by about 29% from access path length one to two, by about 23% from two to three, and by about 18% from three to four. This shows that even for those apps for which no memory exhaustion occurs, an increased access path can significantly increase the time and memory required for the analysis. From this observation, we can draw the conclusion that for maximum scalability, lower access path lengths are, in the average case, the better choice.

However, the reader be reminded that the magnitude of the observed increase in analysis time for longer access paths is mainly due to certain apps in which the issue of over-tainting on short access paths is not significant, but for which many different tainted access paths exists when scaling the access path length. When scaling the access path length from one to two, for example, only 14 out of 25 apps experience an increase in computation time, while for 9 apps the time decreases. For only four apps out of the 14, the required analysis time increased by more than 20% which is acceptable given the gain in precision. When looking at the memory consumption, it increases for 18 apps out of 25 when moving from access path length one to two. For 6 apps, it decreases. Only 10 apps out of the 18 experience a growth in memory consumption that exceeds 20%. Figure 22(a) shows the time required for analyzing the apps from our experimental set (y axis) over the configured maximum access path length (x axis). This chart compares the general time intervals in which the app analyses terminate over the various access path lengths rather than the effect of increasing the access path length for a single app. Blue circles represent apps for which the analysis terminates successfully. Red triangles represent analyses that were aborted early because of memory exhaustion. We denote the average analysis times with black squares. Figure 22(b) shows a similar plot for memory consumption (x axis) over access path length (y axis). From the figures, we can again conclude that increasing the maximum access path length increases the risk of memory exhaustion. When moving from left to right in the plots, the number of red triangles increases. Those apps that lead to the upper blue does for access path length one produce red triangles for length four. This means that analyzing some apps, for which the analysis could be completed with a short access path length, is infeasible for longer access paths.

When only considering the distribution of the blue dots, i.e., the values for those apps for which the analysis terminates, we can conclude that the stack of dots becomes narrower. This mainly happens because the upper blue does, i.e., those apps for which the analysis already took longer and/or required more memory with a small access path length, are turned into red triangles, i.e., the analysis can no longer be completed, when further increasing the access path length. For those apps for which the analysis completed quickly with a short access path length, one would, on the other hand, expect the range of blue dots to move up, because the average time and memory requirements increase with longer access paths as reported earlier. This effect is, however, not visible in the plots due to the broad range the values already have depending on the particular app. Keep in mind that the scales of the time plot are in thousands of seconds to capture the outliers. Most analyses, however, terminate in less than 150 seconds for access path length one as we have shown in Table 11. A similar reasoning can be applied to the memory plot. We also plot the same number using error bars in Figure 22(c) for time consumption and Figure 22(d) for memory consumption. These two plots lead to similar insights. The blue dot on the bars shows the average values for time and memory consumption. Note that the average also includes the outliers for which the analysis has become infeasible. Given the scale the values already have depending on the app, the absolute increase of the computation time and memory consumption in seconds and megabytes, respectively, is no longer significant in comparison to the outliers. Even an average increase of 29% from length one to two is barely visible for an app with a low baseline given that other apps take almost a hundred times as long even in the base case. Therefore, the plot only shows only a slight trend upward. From access path length two onward, the memory exhaustion cases are also mainly responsible for the height of the error bars, which becomes evident when comparing the error bars to the respective scatter plots above. Since the average time and memory consumption is very close to the respective

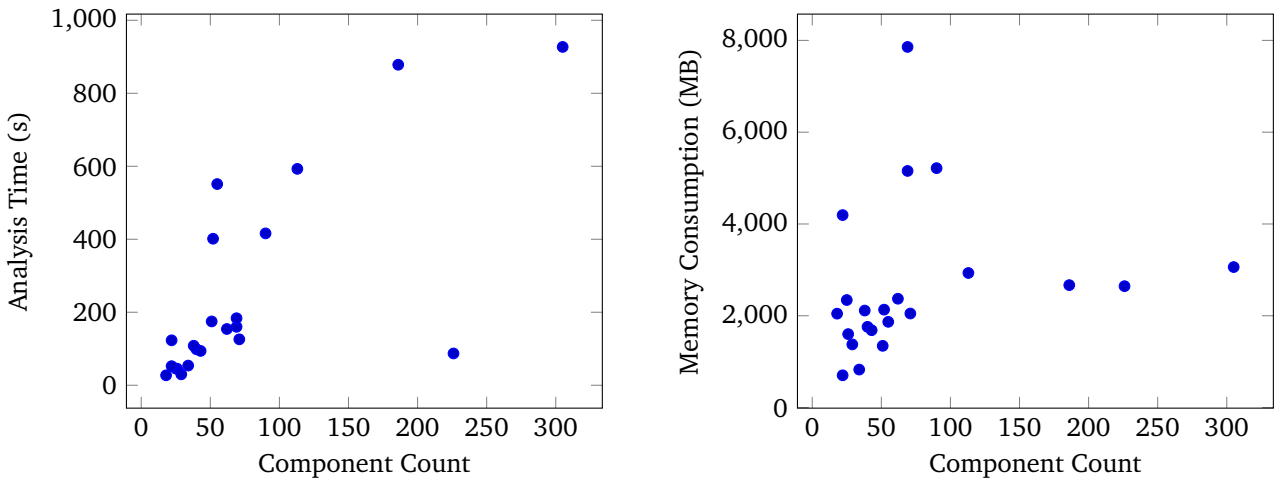
minimum values, we can conclude that analyzing most apps is feasible even with larger access path lengths in FlowDroid. This is no contradiction to the notable increase in the average numbers which is due to certain apps that already start with a larger baseline for access path length one.
In more detail, we observe that the computation time does indeed increase sharply for *some* apps such as the Facebook app. From access path length one to two, it rises by about 480%, and from length two to three again by about 23%. The CNN app shows a similar picture with a rise by 118% from access path length one to length two, and by 292% from length two to length three. On the other hand, many other apps experience a drop in computation time such as the AutoScout24 app, whose analysis time decreases by about 47% when moving from access path length one to two. For yet other apps, the analysis time stays approximately the same, even when increasing the access path length to four or five. The observation that the increase in computation time is not evenly distributed across the apps and some apps even show the inverse effect (i.e., a decrease in analysis time), explains why the overall distribution of time and memory depicted in the scatter plots does not change significantly. Also recall that the app that took the longest to analyze (Instagram) did not finish in reasonable time for access paths longer than one and was thus excluded afterward. In Figure 22(a), this means that the upper left dot does not relocate, but disappear. Further note that we had to set a maximum heap size, which was 100 GB in our experiments. We did not further increase the maximum heap size if the analysis failed with 100 GB, but rather reported the actual time and memory consumption till the analysis was interrupted. Consequently, the values for the red triangles, i.e., the memory exhaustion cases, do not correctly reflect the time and memory that would be necessary to fully conduct the analysis given that an unlimited maximum space (or simply a sufficiently large amount of heap space) were available. Therefore, the precise data points of the red triangles might be misleading, though the fact that their number increases, is a very clear sign that increasing the access path length has a negative impact on scalability, and may lead to more aborts due to memory exhaustion.
Similar to our base measurements in Section 8.2, there is no simple a-priori estimate of how the analysis of a particular app will behave when increasing the access path length. Code size or number of sources are, again, not directly correlated to the increase in time and memory consumption. For an analyst, who needs maximum precision, we therefore propose to run the analysis using a long maximum length and supply a timeout. If the analysis does not complete within this time frame, he can restart it with a lower maximum length. Note again that even when the analysis is aborted due to memory exhaustion or a timeout, the subset of results computed so far is still reported.

8.4 One Component at a Time

As explained in Section 5.4.3, FlowDroid normally builds one dummy main method for the entire app, which can lead to a large callgraph and to memory exhaustion during taint propagation. To circumvent the problem, the app can be segmented such that only one component is analyzed at a time. While this reduces overall memory pressure, it also means that the general initialization overhead that is required before each taint analysis must be invested not only once, but for each component, which can also increase the time required to analyze the app. In this section, we measure the effect of analyzing one component at a time on both time and memory consumption on our set of sample apps defined in Section 8.2. Table 12 shows the results of our measurements. We repeat the base case data, i.e., the time and memory required for analyzing the apps in our test set with all components together. We contrast these numbers with the measurements for running the analysis with one component at a time.
Only analyzing one component at a time can significantly reduce the memory consumption of the analysis. For the Instagram app, the required amount of memory was reduced by more than 96%. At the same time, the analysis took about 59% longer, though. If the originally required about 50 GB of memory are not availabe, this tradeoff is helpful. One can, however, not conclude that with this mode, all apps can be analyzed. For some of the apps, for which we experienced memory exhaustion during our base evaluation described in Section 8.2, the analysis now no longer runs out of memory, but out of time instead. According to the table, this happened for the eBay and LinkedIn apps. When analyzing those apps using the one-component-at-a-time mode, the analysis did not complete within a reasonable time frame. After more than one hour, we stopped it. Therefore, we do not report any time or memory values for these apps. It becomes apparent, that analyzing one component at a time is a tradeoff between time and memory for such apps. The memory issue has now been shifted to a time issue. While it might not be possible to supply more memory, the analyst can chose to supply significantly more time to complete the analysis nevertheless, assuming that it does eventually terminate. For the Amazon Kindle app, for which the analysis did complete when analyzing all components together, we stopped the analysis after more than one hour, when it had only processed 61 of the 85 components inside the app. In this case, the tradeoff was negative, i.e., analyzing only

App	Normal / Base Case			One Component At A Time		
	Runtime (s)	Memory (MB)	Completed	Runtime (s)	Memory (MB)	Completed
Adobe Reader	15.15	1,765.29	Yes	45.62	2,342.94	Yes
Amazon Kindle	281.15	14,850.77	Yes	-	-	No
Angry Birds	29.24	3,642.33	Yes	45.31	1,600.73	Yes
AutoScout24	79.66	4,839.29	Yes	94.01	1,685.42	Yes
BurgerKing	30.64	4,157.71	Yes	108.27	2,113.61	Yes
CCN	44.55	3,197.52	Yes	160.08	7,857.36	Yes
eBay	1,079.66	78,507.27	No	-	-	No
Facebook	142.54	3,143.53	Yes	878.16	2,667.62	Yes
Facebook Messenger	24.08	1,997.73	Yes	926.96	3,061.05	Yes
Google Plus	40.99	5,033.85	Yes	87.09	2,644.85	Yes
Instagram	1,352.12	49,322.62	Yes	551.22	1,867.85	Yes
LinkedIn	561.72	74,199.98	No	-	-	No
Microsoft Outlook	51.41	4,969.30	Yes	416.05	5,216.46	Yes
Microsoft Word	49.89	7,656.06	Yes	98.53	1,759.81	Yes
Netflix	90.71	7,536.65	Yes	174.81	1,344.93	Yes
Offi Journey Planner	19.39	2,044.04	Yes	52.27	702.88	Yes
Opera	32.83	2,935.80	Yes	154.09	2,371.07	Yes
PayPal	87.24	5,461.64	Yes	592.89	2,932.15	Yes
Pinterest	130.35	5,383.73	Yes	401.43	2,133.48	Yes
Pokemon GO	39.17	5,422.42	Yes	27.22	2,045.81	Yes
Skype	27.67	808.71	Yes	125.85	2,049.01	Yes
Telekom Mail	25.51	2,742.11	Yes	53.86	827.86	Yes
Tinder	47.63	1,170.53	Yes	123.11	4,193.73	Yes
VLC	16.09	2,142.38	Yes	29.86	1,376.05	Yes
Wetter.com	56.75	4,921.25	Yes	183.64	5,155.71	Yes

Table 12: FlowDroid Performance in One Component At A Time Mode



(a) Component Count vs. Analysis Time

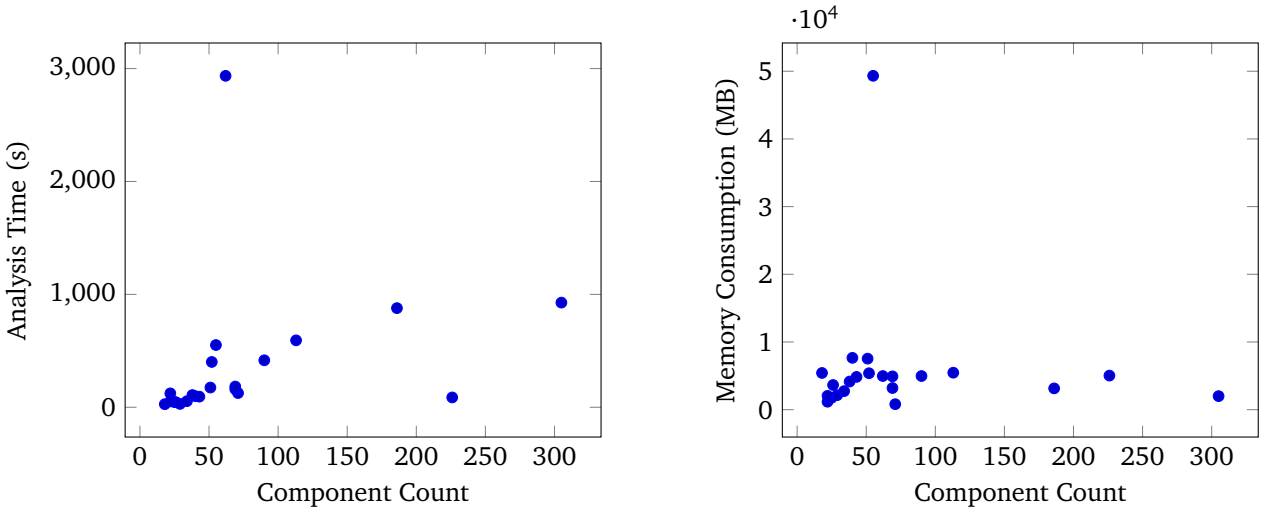
(b) Component Count vs. Memory Consumption

Figure 23: The Effect of the One-Component-At-A-Time Mode

one component at made a previously possible analysis infeasible. Therefore, this mode should only be used as a second chance if the analysis is not possible in normal mode.
On average over all apps, the computation time increases by 374%, while memory consumption drops by 9%. On average, the apps had 79 components. Out of the 25 apps, the time improved for two apps, and was increased for 20 apps. For three apps, we could not obtain time and memory values, because the analysis did not complete. Memory consumption is more likely to decrease as this happens for 15 apps out of 25. Only for seven apps, the memory consumption increases. If we only take those apps into account for which the memory consumption decreases, this decrease is by about 57%. This result is coherent with our conclusion that for some apps, analyzing one component at a time can significantly improve the scalability of FlowDroid. Nevertheless, it is not generally recommended and can have negative efforts for those apps for which it is not necessary, which is also why the average memory savings are much lower when also incuding those apps for which the mode is not helpful.
In Figure 23, we correlate the time and memory required for the analysis with the number of components in the app. Since one run is necessary per component, the time obvisouly increases with the number of components as shown in Figure 23(a), although this is only a trend and accepts outliers. For the memory consumption, the number of components is not relevant as shown in Figure 23(b) and as expected, because each component is analyzed in isolation. For comparison, Figure 24 shows the correlation between component count and time/memory consumption in the normal mode, i.e., when all components are analyzed together. We can see that normally, there is no such evident correlation between component count and analysis time. However, one cannot find a strict correlation between component count and memory consumption either, making an a-priori choice between normal and one-component-at-a-time mode complicated.

8.5 Omitting Android Callbacks

In some cases, analysts need to run the data flow tracker in highly memory-constrained environments. In such cases, it is acceptable to miss potential leaks in favor of performance and scalability. The results of such a quick analysis are usually used to gain a first impression of how the app deals with privacy-sensitive information. FlowDroid offers an option to ignore the Android callbacks completely for such use cases. The analysis then runs on an incomplete callgraph which misses all edges that would go into callback methods in the full analysis. On the other hand, this saves the effort required to collect the callbacks from inside the app, and also greatly reduces the amount of code that needs to be analyzed. While this tradeoff may seem arbitrary and crude from a scientific point of view, we found that is used in practice when actual users configure the data flow solver and need to run it on machines with as little as 2 GB of total main memory, in which not only FlowDroid, but also the operating system must fit.
Table 13 compares the time and memory consumption of running the analysis with the Android callbacks included to the respective values when running the analysis without callback support. We can observe that the time required for the analysis decreases by 48% on average when disabling callbacks. At the same time, the number of found



(a) Component Count vs. Original Analysis Time

(b) Component Count vs. Original Memory Consumption

Figure 24: Original Performance vs. One-Component-At-A-Time Mode

App	With Callbacks			Without Callbacks		
	Runtime (s)	Memory (MB)	Leaks	Runtime (s)	Memory (MB)	Leaks
Adobe Reader	15.15	1,765.29	39	8.02	1,458.79	31
Amazon Kindle	281.15	14,850.77	99	258.30	12,242.64	92
Angry Birds	29.24	3,642.33	39	20.97	3,636.69	29
AutoScout24	79.66	4,839.29	43	23.68	4,522.21	15
BurgerKing	30.64	4,157.71	32	20.80	4,090.44	19
CCN	44.55	3,197.52	184	20.21	3,078.26	86
eBay	1,079.66	78,507.27	269	836.54	76,424.57	105
Facebook	142.54	3,143.53	154	81.54	9,395.24	104
Facebook Messenger	24.08	1,997.73	14	8.89	1,412.12	1
Google Plus	40.99	5,033.85	1	28.19	4,968.59	1
Instagram	1,352.12	49,322.62	132	421.80	41,635.33	104
LinkedIn	561.72	74,199.98	131	357.90	74,991.60	63
Microsoft Outlook	51.41	4,969.30	57	23.67	4,404.54	37
Microsoft Word	49.89	7,656.06	26	21.21	4,406.22	14
Netflix	90.71	7,536.65	74	60.86	7,134.28	52
Offi Journey Planner	19.39	2,044.04	13	9.30	1,721.83	3
Opera	32.83	2,935.80	57	17.75	2,734.55	57
PayPal	87.24	5,461.64	90	33.34	4,928.26	56
Pinterest	130.35	5,383.73	107	36.73	4,801.78	83
Pokemon GO	39.17	5,422.42	20	12.14	2,847.63	8
Skype	27.67	808.71	21	13.82	2,635.96	13
Telekom Mail	25.51	2,742.11	66	13.36	2,648.76	59
Tinder	47.63	1,170.53	61	18.75	3,718.93	26
VLC	16.09	2,142.38	15	7.85	2,010.93	11
Wetter.com	56.75	4,921.25	85	31.68	4,497.04	73

Table 13: FlowDroid Performance With and Without Callbacks

leaks also decrases by 37%. This is expected, because analyzing less code takes less time, but can only discover those leaks that are present in the subset of the code that is still being analyzed.
For the memory consumption, the same reduction effect we see for the analysis time cannot be observed. Contrary to what one would expect from theory, the memory consumption *increases* by about 15% when disabling callbacks. The most likely explanation is that this is an artifact of how we measure the memory consumption. After each of the ten analysis runs has completed, and before resetting the references to the solver objects to `null`, we obtain the amount of heap space used in the JVM. If the analysis only needs to work on a small target app, because all code only reachable through callbacks is excluded, this puts less pressure on the garbage collector during the analysis. Consequently, less frequent runs are performed to free up memory. In the end, more heap space is in use than strictly necessary, simply because there is no need to free it up. Note that explicitly calling `Runtime.gc()` does not necessarily lead to full gargabe collection either, as the times when garbage collection is actually triggered and the settings with which it is run are defined by the JVM rather than the user code. Note that for the eBay app and the LinkedIn app, the analysis was interrupted early due to memory exhaustion when including the callbacks. This still happens even when disabling callbacks, which again shows that code size alone is not a good indicator of the time and memory required for the analysis. The code size reduction was substantial for both the eBay and the LinkedIn app when excluding the callbacks, but still did not allow the analysis to complete.
In summary, disabling callbacks is an efficient technique if one wants to get a quick first overview over the app and does not necessarily need to find all the leaks in this first step. While we cannot conclude that excluding callbacks from the analysis also helps reduce the memory consumption, it might as well be the case once the JVM gets under memory pressure and needs to perform more aggressive garbage collection. In the case of the Instagram app, for instance, the memory consumption dropped by about 8 GB, and by about 2 GB for the Amazon Kindle app. For the smaller apps and those apps that experience complete memory exhaustion, we do not see those effects, though. In the latter case, we might observe a reduced memory consumption when giving the analysis sufficient memory to complete (i.e., more than 100 GB) in the first place.

8.6 Disabling Data Type Propagation

In Section 4.11, we presented propagating type information along with the taint abstractions to compensate imprecisions caused by the context-insensitive SPARK callgraph algorithm. In the IFDS call flow functions this propagated type information can be used to prune invalid callees and kill the incoming taints so that only those callees are actually processed with which the type information from the taint abstraction is compatible. In our base configuration shown in Section 8.1, this extra propagation and checking is enabled which is also the FlowDroid default. In Table 14, we compare the time and memory requirements of this base case to the respective values when disabling the type propagation feature. We also show the impact on the number of leaks detected by the analysis.
On average, disabling data type propagation decreases the runtime by about 17% and the memory consumption by about 4%. This result happens because of the reduced re-usability of IFDS method summaries that we already discussed in Section 4.11. On the other hand, disabling the type checking increases the number of detected leaks by about 11%, which is also expected. While these numbers show a general trend that type propagation and checking trades time and memory for precision, this trend does not apply to each single app. Analyzing the Facebook app, for instance, takes almost 84% longer and requires more than 194% more memory without the type checking, in addition to 16% more leaks being detected. In this case, the spurious taint propagations caused by the callgraph imprecision greatly outweigh the reduced applicability of the IFDS summaries. The Pokemon GO app exhibits the inverse behavior. In that case, propagating types requires more than 60% more time and almost 32% more memory, while there the number of leaks does not change, i.e., the extra type information does not increase the analysis precision at all. The eBay app is a special case. Without data type propagation, the number of detected leaks increases by about 82% to 489. Additionally, the resulting taint propagation graph is so large that the path reconstruction step times out. Therefore, we cannot report the full time and memory consumption for the overall analysis. In other words, type propagation and checking is required to even make this analysis feasible. Additionally, since the type checks help improve the precision in the average case, and the extra time and memory requirement is sufficiently low, we chose to enable this feature by default.

8.7 Fast (But Imprecise) Callback Collection

FlowDroid support two different algorithms for collecting the callbacks that an app registers with the Android operating system. The first technique, described in Section 5.3.4, starts with a dummy main method containing only the lifecycle methods of the components registered in the manifest file, and them iteratively extends this model.

App	With Type Propagation			Without Type Propagation		
	Runtime (s)	Memory (MB)	Leaks	Runtime (s)	Memory (MB)	Leaks
Adobe Reader	15.15	1,765.29	39	13.93	1,583.39	40
Amazon Kindle	281.15	14,850.77	99	243.54	10,214.25	106
Angry Birds	29.24	3,642.33	39	28.64	1,774.49	39
AutoScout24	79.66	4,839.29	43	36.11	2,214.73	44
BurgerKing	30.64	4,157.71	32	20.95	1,806.05	32
CCN	44.55	3,197.52	184	46.80	5,104.99	220
eBay	1,079.66	78,507.27	269	Timeout	Timeout	489
Facebook	142.54	3,143.53	154	262.25	9,250.56	179
Facebook Messenger	24.08	1,997.73	14	20.18	1,932.82	14
Google Plus	40.99	5,033.85	1	41.86	1,838.67	1
Instagram	1,352.12	49,322.62	132	89.48	4,690.35	133
LinkedIn	561.72	74,199.98	131	Timeout	Timeout	Timeout
Microsoft Outlook	51.41	4,969.30	57	54.32	2,177.76	61
Microsoft Word	49.89	7,656.06	26	31.01	1,930.62	26
Netflix	90.71	7,536.65	74	52.80	3,696.38	81
Offi Journey Planner	19.39	2,044.04	13	20.86	2,601.43	16
Opera	32.83	2,935.80	57	33.09	2,818.93	58
PayPal	87.24	5,461.64	90	64.63	3,153.94	93
Pinterest	130.35	5,383.73	107	106.01	3,626.70	109
Pokemon GO	39.17	5,422.42	20	15.55	3,696.38	20
Skype	27.67	808.71	21	21.08	3,174.29	21
Telekom Mail	25.51	2,742.11	66	23.01	2,434.20	66
Tinder	47.63	1,170.53	61	38.89	1,928.18	105
VLC	16.09	2,142.38	15	10.77	1,832.30	17
Wetter.com	56.75	4,921.25	85	53.84	2,198.31	86

Table 14: FlowDroid Performance With and Without Type Propagation

App	Precise Callbacks			Approximate Callbacks		
	Runtime (s)	Memory (MB)	Leaks	Runtime (s)	Memory (MB)	Leaks
Adobe Reader	15.15	1,765.29	39	28.67	3,009.56	32
Amazon Kindle	281.15	14,850.77	99	259.58	18,230.13	92
Angry Birds	29.24	3,642.33	39	40.04	7,395.32	32
AutoScout24	79.66	4,839.29	43	50.33	5,253.60	24
BurgerKing	30.64	4,157.71	32	36.46	4,387.80	19
CCN	44.55	3,197.52	184	92.54	7,602.92	101
eBay	1,079.66	78,507.27	269	3,911.44	62,621.80	191
Facebook	142.54	3,143.53	154	29,633.57	24,626.19	120
Facebook Messenger	24.08	1,997.73	14	17.37	2,477.77	10
Google Plus	40.99	5,033.85	1	63.83	5,227.04	1
Instagram	1,352.12	49,322.62	132	446.27	39,037.63	98
LinkedIn	561.72	74,199.98	131	788.12	77,033.43	73
Microsoft Outlook	51.41	4,969.30	57	44.53	3,896.04	39
Microsoft Word	49.89	7,656.06	26	79.68	7,366.44	15
Netflix	90.71	7,536.65	74	67.86	5,748.64	54
Offi Journey Planner	19.39	2,044.04	13	17.58	1,751.14	7
Opera	32.83	2,935.80	57	31.19	4,953.32	52
PayPal	87.24	5,461.64	90	56.71	5,722.23	56
Pinterest	130.35	5,383.73	107	60.57	4,125.49	92
Pokemon GO	39.17	5,422.42	20	25.00	6,469.41	8
Skype	27.67	808.71	21	17.37	2,477.77	10
Telekom Mail	25.51	2,742.11	66	19.77	3,006.23	61
Tinder	47.63	1,170.53	61	40.08	4,503.92	29
VLC	16.09	2,142.38	15	10.88	2,000.52	12
Wetter.com	56.75	4,921.25	85	43.66	4,260.36	75

Table 15: Performance of Precise vs. Approximate Callback Collection

When new callbacks have been found (i.e., call sites registering new callbacks are reachable from the current version of the dummy main method), a new dummy main method is constructed, and the analysis is repeated until a fixed point has been reached. This technique yields a precise mapping between components and their respective callbacks, because FlowDroid can capture from which component's lifecycle the callback registration is transitively reachable. On the downside, this process takes several iterations and callgraph re-computations for real-world apps. As an alternative, FlowDroid can over-approximate the callbacks by simply iterating over all call sites that register callbacks in the whole app without computing any form of reachability analysis as explained in Section 5.3.5. In this case, all callbacks are assumed to be valid for all components. This is less precise, but reduces the time required for the callback analysis.

Table 15 shows our performance evaluation that compares the two modes on our set of real-world apps. We note that, depending on the app, either mode can be more efficient than the other. For some apps, the time saved during callback analysis when switching to the imprecise mode is much less than the extra time that needs to be spent on spurious taint propagation later on as a result of the imprecise app model. For those apps, however, for which the imprecise model has no significant impact on performance, the imprecise, but faster callback collection can be beneficial. In the worst case of our test set, the Facebook app, the imprecise model lead to more than 200 times the computation time that was required with the precise model. Memory consumption increased by 6 times due to the imprecision. When removing this outlier, the time total analysis required with the imprecise model was about 9% than with the precise one and required about 34% more memory. In the average case, chosing the precise model is, therefore, considered to be the best choice. For 16 out of our 25 apps, however, the imprecise mode was faster, and in 9 cases, it consumed less memory than the precise mode. In the cases in which the imprecise mode was faster, it saved about 28% of the computation time on average. For the cases in which the memory consumption was reduced, it saved about 16%. We note that there is no clear correlation between the number of components inside an app and the impact of using one callgraph collection algorithm over the other, neither for performance, nor for memory consumption.

Another important consideration when chosing a callgraph collection algorithm is the precision and recall of the overall analysis. One might assume that chosing a less precise association between callbacks and their hosting components can only lead to more leaks, because every callback is associated with every component. Table 15, however, shows that this is not the case. Instead, taints get overwritten or get placed into the wrong fields, effectively preventing the true leaks from being detected. On average, FlowDroid detects 29% *less* leaks with the imprecise callback analysis. For the LinkedIn and eBay apps, the number of leaks is approximate, because for these apps, the analysis runs out of memory with either callback analysis algorithm. More precisely: it is aborted by FlowDroid's memory thresholding technique (see Section 4.12.3) before it can finish the analysis. Therefore, all values reported for these two apps only reflect partial results.

8.8 Flow-Insensitive Data Flow Solver

The FlowDroid data flow tracker is formulated as an IFDS problem such that the overall analysis is flow-, context-, field-, and object-sensitive. As described in Section 4.12.4, there is, however, also a variant of the solver that can process the same IFDS problem, but that is flow-insensitive. In this section, we evaluate the performance and memory consumption of this solver variant. Without flow-sensitivity, we already assume that the precision degrades and, consequently, more leaks are found. This assumption is backed by the experimental data in Table 16. On average, the flow-insensitive solver finds about 134% more leaks than the flow-sensitive one. For some apps such as the Telekom Mail app, the increase is by more than 340%, for the Facebook Messenger app even by more than 690% (up to 111 leaks from 14 with the flow-sensitive solver). In no case, we saw a decrease in the number of leaks found. Technically, a flow-insensitive analysis must propagate all taints that are created somewhere in a method over all statements in that method, leading to more flow function evaluations. This not only increased the risk of false positives, but also requires more time. On average, the flow-insensitive analysis takes about 180% longer than the flow-sensitive one and consumes about 197% more memory. 6 out of 25 apps cannot even be analyzed anymore due to timeouts. We find that in most of these cases (4 apps), the timeout happens during the path reconstruction step and is caused by the large increase in the number of results that need to be processed. For these cases, we can still report the number of discovered leaks in Table 16, though no performance and memory data is available. For smaller and less complicated apps, this increase can still be handled, but for the large and complex apps, there are too many possible paths through the the taint graph of the app from the sink back to potential sources.
It is important to note that this large increase in time and memory consumption is mainly due to three outliers: Telekom Mail, Oeffi Public Transport Planner, and Facebook Messenger. For these apps, the time increases by more than 400%, and memory consumption by more than 600% for each app. If we remove these outliers, the average increase in time consumption is by only 12.7% and the average increase in memory consumption drops to 72%. The increase in time and memory does not apply to every single app, either. In 6 cases, the flow-insensitive analysis was faster than the flow-sensitive one. In this case, the time reduction was by 31% on average. The memory consumption decrased for 9 apps, with an average decrease by 39% in those cases. In total, using the flow-insensitive data flow solver should not be the default configuration. Unfortunately, we cannot confirm that this configuration can help analyze apps for which the analysis is infeasible with the default configuration, either.

8.9 Comparing FastSolver And The Heros Data Flow Solver

As explained in Section 4.12, FlowDroid uses a custom, highly optimized IFDS solver called the *FastSolver* by default. To allow for a comparison between this solver and Heros [23], which his the default IDE and IFDS solver commonly used with Soot, FlowDroid abstracts away from the concrete solver using an interface and provides implementations that integrate both Heros and FastSolver. In this section, we compare the performance and memory consumption of the two solvers on our set of real-world test apps.
Aside from the differences in performance and memory consumption, we also notice that the analysis yields more leaks (99.6% more on average) when when it is run with the Heros solver in comparison to FastSolver. For the VLC media player, the number of leaks increases from 15 to 59, an increase by almost 300%. This is mainly because Heros does not support any filtering techniques such as the overtaint filtering presented in Section 5.4.1. In short, if a full Android component is tainted when running the analysis with the Heros solver, this leads to a large number of spurious taints. Propagating more taints also consumes more time and memory. On average, the analysis time increases by more than 360% and the required memory by more than 100%. There was only a single app (out of 25) for which the analysis was faster with the Heros solver. For the AutoScout24 app, the Heros solver required around 40% less time and around 48% less memory than FlowDroid's FastSolver. For all other

App	Flow-Sensitive Solver			Flow-Insensitive Solver		
	Runtime (s)	Memory (MB)	Leaks	Runtime (s)	Memory (MB)	Leaks
Adobe Reader	15.15	1,765.29	39	16.06	1,964.37	63
Amazon Kindle	281.15	14,850.77	99	Timeout	Timeout	191
Angry Birds	29.24	3,642.33	39	34.89	2,992.23	56
AutoScout24	79.66	4,839.29	43	34.77	2,210.81	86
BurgerKing	30.64	4,157.71	32	24.08	2,081.57	59
CCN	44.55	3,197.52	184	73.27	7,112.03	475
eBay	1,079.66	78,507.27	269	Timeout	Timeout	-
Facebook	142.54	3,143.53	154	Timeout	Timeout	283
Facebook Messenger	24.08	1,997.73	14	140.35	15,592.10	111
Google Plus	40.99	5,033.85	1	41.02	1,795.33	3
Instagram	1,352.12	49,322.62	132	Timeout	Timeout	267
LinkedIn	561.72	74,199.98	131	Timeout	Timeout	-
Microsoft Outlook	51.41	4,969.30	57	59.96	4,311.79	139
Microsoft Word	49.89	7,656.06	26	35.22	2,567.32	52
Netflix	90.71	7,536.65	74	103.81	8,483.13	148
Offi Journey Planner	19.39	2,044.04	13	224.49	14,485.18	20
Opera	32.83	2,935.80	57	50.95	8,374.45	124
PayPal	87.24	5,461.64	90	Timeout	Timeout	152
Pinterest	130.35	5,383.73	107	331.92	27,970.23	199
Pokemon GO	39.17	5,422.42	20	19.88	2,292.39	34
Skype	27.67	808.71	21	26.03	3,041.16	32
Telekom Mail	25.51	2,742.11	66	453.19	38,690.05	292
Tinder	47.63	1,170.53	61	70.68	6,776.71	155
VLC	16.09	2,142.38	15	12.38	1,934.47	32
Wetter.com	56.75	4,921.25	85	61.96	3,922.93	142

Table 16: Performance of The Flow-Insensitive Solver Variant

App	FlowDroid's FastSolver			Heros		
	Runtime (s)	Memory (MB)	Leaks	Runtime (s)	Memory (MB)	Leaks
Adobe Reader	15.15	1,765.29	39	46.34	3,361.11	87
Amazon Kindle	281.15	14,850.77	99	3,649.07	50,230.77	125
Angry Birds	29.24	3,642.33	39	52.26	3,279.24	48
AutoScout24	79.66	4,839.29	43	47.73	2,540.20	78
BurgerKing	30.64	4,157.71	32	31.23	2,245.03	41
CCN	44.55	3,197.52	184	410.03	9,851.06	390
eBay	1,079.66	78,507.27	269	Timeout	Timeout	-
Facebook	142.54	3,143.53	154	Timeout	Timeout	-
Facebook Messenger	24.08	1,997.73	14	260.65	4,127.31	38
Google Plus	40.99	5,033.85	1	41.04	1,737.18	1
Instagram	1,352.12	49,322.62	132	Timeout	Timeout	-
LinkedIn	561.72	74,199.98	131	Timeout	Timeout	-
Microsoft Outlook	51.41	4,969.30	57	180.67	6,622.72	128
Microsoft Word	49.89	7,656.06	26	56.41	3,668.68	55
Netflix	90.71	7,536.65	74	Timeout	Timeout	-
Offi Journey Planner	19.39	2,044.04	13	315.63	7,167.98	13
Opera	32.83	2,935.80	57	115.55	4,236.37	96
PayPal	87.24	5,461.64	90	331.68	9,206.89	180
Pinterest	130.35	5,383.73	107	395.71	9,906.30	216
Pokemon GO	39.17	5,422.42	20	50.71	3,886.44	21
Skype	27.67	808.71	21	218.55	6,549.40	69
Telekom Mail	25.51	2,742.11	66	43.20	3,048.28	197
Tinder	47.63	1,170.53	61	168.45	6,284.33	133
VLC	16.09	2,142.38	15	49.99	3,205.15	59
Wetter.com	56.75	4,921.25	85	166.21	5,133.05	150

Table 17: Performance Comparison Between FastSolver and Heros

apps, the performance of the FastSolver was better. When considering only the memory consumption, the Heros solver required less memory for 6 out of the 25 apps in our test set. The impact these apps have on the average is, however, greatly outweighed by those apps for which Heros consumed several hundred percent more memory than the FastSolver (up to around 710% more memory for the Facebook app). In summary, FlowDroid's FastSolver substantially improves the performance of the data flow analysis in the average case in comparison to existing off-the-shelf solvers such as Heros.

The decreased number of leaks detected by FastSolver in comparison to Heros is potentially caused by the higher precision of FastSolver. To verify this assumption, we re-ran the test cases from the DroidBench micro-benchmark suite (see Section 7) with the Heros solver. For two of the cases (FlowSensitivity1 in the *Aliasing* category, and Ordering1 in the `Callbacks` category), this lead to two additonal false positives (two for FlowSensitivity1 and one for Ordering1). For all other test cases, the results were equivalent. On the overall benchmark suite of 190 apps, this difference in precision is not significant. It is far from the 99.6 percent increase in the number of leaks that we observed on the real-world apps. Therefore, we cannot finally verify our hypothesis that FastSolver is significantly more precise than Heros. Since we do not have a ground truth on the real-world apps, a fair comparison is hard to achieve on this test set.

9 Multi-Platform Static Analysis[41]

The previous sections of this thesis have focused on Android apps. Similar privacy issues, however, also exist on other mobile, embedded and desktop platforms. Generally, they can be solved using the same or at least very similar techniques. Therefore, this section describes an outlook on broadening the scope of the techniques presented in this paper beyond Java and Android. Many of the techniques presented in the academic literature have been designed for particular target platforms and programming languages. Popular targets of research include Java programs and Android apps. In reality, different programs are, however, often written in different programming languages. The choice of the language depends on the requirements of the deployment target, the language skills of the developer, the availability of powerful frameworks, and many other criteria. These criteria often differ significantly from the reasons why platforms or programming languages are chosen for academic research. As a consequence, the focus of research attention on individual platforms has lead to a large divergence in the availability of static-analysis tools for different platforms. This hinders the practical applicability of static-analysis tools.

Some analyses [60, 75] are built directly on top of platfom-specific tools such as disassemblers like Dexdump[42], others [1, 46, 150, 156] use platform-specific frameworks such as AndroGuard [36]. Migrating these tools to other platforms is a major undertaking. The FlowDroid data flow tracker, on the other hand, is designed to be platform-independent and thus to be applicable to all target platforms and programming languages to which Soot, as the framework on which FlowDroid is based, can be applied. As long as the target program can be converted to Jimple code, the FlowDroid engine can serve as a basis for building a static data flow tracker for the respective platform. Building upon such an intermediate representation is a key advantage when applying a tool or a framework to multiple platforms. The intermediate representation provides a significant abstraction of assembly-level opcodes. In the case of Soot, this additional level of abstraction in comparison to, e.g., a disassembler, allows the framework to convert into Jimple not only Java bytecode, but also Android's Dalvik bytecode, through the Dexpler front-end [18], which already allows FlowDroid to work on two platforms instead of one. For the analysis tool, be it FlowDroid or another tool based on Soot, there is little difference as to whether the client code originated from an Android or Java application. To be precise, while the tool needs to deal with the different peculiarities of the Android or Java libraries and frameworks, the tool can be agnostic to the origin of the Jimple code as such. Of course, similar to the work we conducted for Android, the analysis designer must still provide custom implementations of the varios interfaces of FlowDroid where necessary, i.e., where the default implementation based on the Java semantics is insufficient. Also on the level of the Soot framework itself, Dalvik has different exception semantics than Java, so a different throw analysis must be provided.

Though Java and Android use different bytecodes, their expressiveness is quite similar. The Android compiler (the `dx` tool) generates a Dalvik `classes.dex` file from Java class files. When doing so, it converts Java's stack-based bytecode language into the register-based Dalvik language, but since it starts from Java bytecode, the expressiveness of the Dalvik language is limited the one of Java. Dalvik does not support any language features that are not already present in Java. Therefore, supporting both Java and Android through the same IR is conceptually simple. Supporting other languages in a shared IR, however, is more challenging, because Jimple was originally designed with the semantics of the Java VM in mind. WALA [141], a framework similar to Soot, on the other hand offers *WALA CAst* (*WALA Common Abstract Syntax Tree*), a cross-language source front-end. This front-end currently supports Java and JavaScript as input. As the name implies, CAst, is, however, an AST-style data structure with support for multiple specialized IRs, and not a single, uniform, strongly typed IR. Therefore, one cannot seamlessly switch the framework to another input language and assume the same analyses to continue working as desired. Due to the nature of the highly dynamic JavaScript language, it remains an open question whether a truly shared IR for Java and JavaScript is actually possible or desirable.

In this section, we propose a first step towards turning Soot into a cross-language, cross-platform static analysis tool that produces the same IR irrespective of the input language. This can then, as future work, serve as a foundation for generalizing FlowDroid beyond Java and Android as target platforms. As explained above, the foundation inside the data flow tracker already exists, but requires support by the Soot framework. We therefore explain how we extended the Soot framework to convert into Jimple also the bytecode language of the Microsoft .net framework and the Mono open-source project. While this bytecode language CIL (Common Intermediate Language) is a fully managed-code language just as Java, we show that CIL code nevertheless has significant differences to Java bytecode. We explain how we model CIL's distinct features in Jimple without extending the Jimple language

[41] Large parts of this Section are taken (directly or with minor modifications) from our 2016 SOAP paper[6]

[42] Dexdump is included with the official Android SDK published by Google.

itself. We chose to avoid extensions to Jimple in order to allow researchers to reuse existing analyses based on Jimple such as FlowDroid without requiring changes or extensions on the language level. We hope that these ideas help further broadening the applicability of existing (also other) static-analysis frameworks to other languages and platforms.

9.1 Code Organization in CIL

In Java, every class gets compiled into its own class file. Multiple class files can be packaged together into one JAR archive. In the CLI (*Common Language Infrastructure*, the component that runs the CIL code), on the other hand, a compiled class itself has no representation in the file system. Instead, the CLI runtime operates with a collection of classes stored in an *assembly* file. While at a first glance assemblies seem conceptually similar to JAR files, they are artifacts in their own right, resembling a logical package on the language level. For instance, visibility levels include an option for assembly-wide visibility for classes, methods, and fields. Code can use reflection to access an assembly and, e.g., list all classes inside it. Assemblies can be signed and used for code security, i.e., by defining that certain code may only be called from code within assembly that was signed with a specific key.
As Java has no notion of assembly visibility, nor has Jimple, our generation of Jimple code from CIL widens assembly-wide visibility (`internal` in C#) to public visibility. While this may impact some specific analyses, it retains compatibility in general. CIL bytecode represents security restrictions such as signatures and permissions as API calls or attributes in the original code. CLI attributes are similar to Java attributes and are translated into Jimple as Tags attached to the appropriate Jimple artifacts. An analysis usually does not need to cover the semantics of such attributes unless it directly targets security problems. In thus case, it would have to precisely model the correct target platform in either case.
Inside an assembly, classes are structured in namespaces, similar to Java's packages. In Jimple, we therefore represent namespaces as packages. C# also allows one to define aliases for namespaces, but these aliases are resolved to their original names by the compiler and are thus not a challenge for the work presented here. Note that CIL bytecode can reference the same class in the same namespace from two different assemblies. This is not possible in Jimple, unless we treat assembly names as namespace prefixes.
Furthermore, .net languages such as C# support *partial classes*. In a partial class, methods and fields can be scattered among multiple source files which get merged into one complete class at compile time. Conflicting definitions lead to a compiler error. It is also possible to only write the signature of a method in one source file and the implementation in another one. Due to the compile-time merging, the front-end presented in this work can, however, be oblivious to this language feature.

9.2 The CIL Type System

Java distinguishes between Objects and numeric/Boolean values, the former of which always use a pass-by-reference semantics, while the latter use pass-by-value. CIL offers a richer model: on top of regular objects it supports also *value types* called *structs* that are passed by value. All structs have an implicit empty constructor and therefore do not need to be initialized explicitly. The primitive data types such as `int` and `float` are system-defined structs which are declared in the system assembly `mscorlib`. Enums are structs as well, merely defining fixed values that can be type-checked by the compiler.

```
1  void test() {
2    int a = 5;
3    calcRef(ref a);
4    Console.WriteLine(a);
5
6    int[] b;
7    calcOut(b);
8    Console.WriteLine(b[1]);
9  }
```

```
10 int calcRef(ref int a) {
11   a = a + 3;
12 }
13
14 int calcOut(out int[] a) {
15   a = new A[3] { 1, 2, 3 };
16 }
```

Listing 51: Calling Conventions in C#

Since Jimple is based on Java, it cannot directly represent this distinction. We represent CIL classes and structs as normal classes in Jimple. Whenever such Jimple classes are used in the code, we, however, need to correctly

emulate the semantics of the calling convention. We achieve this by cloning the objects that correspond to structs before passing them as method parameters (or base objects for virtual calls). The clone is a member-wise deep-clone on structs only. This means that if a struct contains another struct (recall that primitive types are structs as well in CIL), its value is copied recursively. If a struct contains a reference to a class, this reference is kept as-is in the clone, which is in line with the CIL language semantics.
Note that programmers can also overwrite the default calling convention as shown in Listing 51. In line 3 an `int` value (which is a struct and thus normally passed by value) is explicitly passed by reference. In the `test()` method, the new value will thus be available which is why line 4 will output `8`. We model this behavior by not cloning structs when they are explicitly passed by reference. System-defined primitive structs such as `int` are modeled through classes just like any other struct.
CIL further supports the `out` keyword. This can be used in the same place as the `ref` keyword and makes a parameter behave similar to a return value as shown in line 7 of the example. The output line 8 is `1` from the array constructed in the callee `calcOut`. This is different to `ref` where a reference is passed in and the callee can optionally overwrite the object which is then also passed back out to the caller again. In the case of `out`, nothing goes in and the callee defines the value to be returned. It is important to correctly model this behavior. If we assume a pure by-reference semantics as in Java, false positives can occur in analyses such as taint tracking. Furthermore, not handling the `out` keyword can lead to uninitialized variables as shown in the example. However, the `out` keyword cannot directly be represented in Jimple. Just like Java, Jimple only allows a single return type for a method and assumes all parameters to be input data. We solve this challenge by automatically boxing the respective parameters. For each type that is passed as an `out` parameter, we automatically generate a boxing class that stores the actual data in a field. When the call returns, the data is read from the field.
Structs differ from objects also in the way they are allocated. Before calling a method on an object or accessing one of its fields, the object's constructor must be called. For a struct, this is not necessary. When a struct is declared, the runtime automatically allocates the required memory and fills it with zeroes. Semantically, this initializes all fields to the default values of their respective types, i.e., zero for numeric types, and null for references. In Jimple, just like in Java, such implicit initialization does not exist. Therefore, we create explicit calls to the default constructor for each declared struct before any other method code is created. This simulates the runtime's initialization behavior.

9.3 Modeling the CIL Language Features

In this section, we describe some of the distinct language features of the CIL language and how we model them in the Jimple IR. Due to space constraints, we limit ourselves to the most important features. Recall that the goal is re-usability of existing analyses which requires us to avoid changes or extensions to the Jimple IR.

9.3.1 Generics

```
1 void test() {
2   List<A> lst = getList();
3   A a = lst.get(0);
4   bar(a.toString());
5 }
```

Listing 52: Generic Lists in C#

While Java and the .net languages such as C# all support generics, their handling at compile time is fundamentally different. The example in Listing 52 works for both Java and C#. The Java compiler erases generics by reducing them to the closest common supertype. Therefore, the return type `List<Object>` of method `getList()` will be reduced to `List`. The types of local variables are erased, because they are not needed in the bytecode. A static analysis tool such as Soot must reconstruct these local types from the interface types (parameter types, method return types) and the operations performed on the local variables. In the example, it can only infer `java.lang.Object` as the type for variable `a`. Recall that no generic-type information is available for the list, and thus the return type of the `get()` method is `java.lang.Object` as well. Therefore, the call to `toString()` in line 4 can lead to the `toString()` implementation of `java.lang.Object` or any subtype, making the callgraph greatly imprecise.
In CIL, type information is preserved on local variables as well as on generics. From the CIL bytecode, it is immediately apparent that variable `a` is of type `A` and not of type `System.Object` (which is CIL's equivalent to `java.lang.Object`). Therefore, the CIL frontend can simply inject a typecast. This has several advantages. Firstly,

the time-consuming process of type inference is avoided. Secondly, the use of generic types, most commonly collections, does not reduce the callgraph precision in comparison to explicitly-typed specialized classes. In the example, the set of possible callees for the call in line 4 is directly limited to `A.toString()` or overrides in subtypes.
Inside the generic class (the `List` class in the example), the generic types are reduced to base types, similar to the type reduction performed in the Java compiler. If the generic type is a class type, it is reduced to `System.Object`, unless it is explicitly declared to be the subclass of some other, more concrete type. In the latter case, the generic type is reduced to that given supertype. Note that the frontend needs to apply name mangling for generic classes. As generics are explicit in CIL, it is legal to have two different classes with the same name that only differ in the number of type variables. It is, however, not legal to have the same class name and same number of variables multiple times even if the generic types are limited to subclasses of distinct superclasses. This means that only the number of generics is relevant, not any further information about them. The frontend uses this restriction to create unique names for generic classes. In the example, the `List` class with one type variable becomes `List__1`.

```
1  interface IFace<out T> {
2    T get();
3  }
4
5  interface IFace2<in T> {
6    void add(T data);
7  }
8
9  void test() {
10   IFace<String> if_string = factory();
11   IFace<Object> if_object = if_string;
12   Object obj = if_object.get();
13
14   IFace<Object> if2_object = factory2();
15   IFace2<String> if2_string = if_object;
16   if2_string.add(new Object());
17 }
```

Listing 53: Co- and Contravariance in C#

CIL also allows covariance and contravariance on generic classes. In the example in Listing 53, the interface `IFace` declares an `out` type variable. The `out` keyword indicates that this variable may only be used in place of return types or as `out`-parameter types of methods. This restriction makes it safe to broaden the type through covariance as shown in line 11. Attempting to use the generic parameter `T` as in `in` parameter in the interface leads to a compiler-time error complaining about unsafe covariance. Our frontend assumes that all bytecode to be processed passes type checking. Recall that we generate typecasts to map generic types to actual ones. The type of the generic interface is, just as in Java, independent of the concrete instances of any type variables. Handling covariance is therefore trivial. The frontend only needs to downcast the return type. Similarly to covariance, CIL also allows contravariance on assigments as shown in line 15 in Listing 53. The interface `IFace2` uses an `in`-type variable `T`, which is restricted to incoming parameters of method calls. Trying to use it for `out` parameters or return values of methods will cause type checking to fail. Under the assumption that all code processed by the frontend type checks, this is also handled through an implicit downcast similar to the case of covariance.
Similar to generic classes, CIL also supports generic methods. As with classes, the generic-type information is persisted in the bytecode. The frontend uses thr same name mangling technique as with classes to distinguish overloads with the same name, but different number of generic type arguments. The original generic method is reduced to concrete base types for which typecasts are inserted before and after calls to the method.

9.3.2 Operator Overloading

The .net languages such as C# support operator overloading. In the CIL code, custom operator implementations are treated as normal function calls which makes them easy for Soot to handle. Neither the client analysis nor the CIL front-end need to provide special treatment for this language construct.

9.3.3 Properties

```
1 class TestClass() {
2   private int m_id;
3   public int id {
4     get { return m_id };
5     set { m_id = value; }
6   }
7   public String data { get; set; }
8 }
```

Listing 54: Properties in C#

The .net languages support properties as shown in Listing 54. Properties are used with the same syntax as fields when storing or retrieving values, but are conceptually similar to getter and setter methods. In fact, the compiler automatically converts the property code into methods. The property in line 3 in Listing 54 is an explicit property definition, similar to writing getter and setter methods in Java. Due to this automatic conversion, the frontend does not need any special handling for properties and can simply keep the method invocations generated by the compiler.
C# also supports a simplified syntax for properties that do not need any additional code, and only store a value. Line 7 declares such a property. The compiler automatically generates a private field for it, as well as getter and setter methods that store and retrieve the value for this field. The implications for the frontend are the same as for explicit properties. The handling of *indexers* is similar to properties. In C#, an indexer is a property that can be accessed with an index, similar to an array. In the CIL bytecode, this is translated into getter and setter methods that take the index as a parameter.

9.3.4 Delegates

```
1  class TestClass {
2    delegate void MyDelegate(String inStr);
3
4    void test() {
5      MyDelegate md = delegate (String inStr) {
6        Console.WriteLine(inStr);
7      };
8      md += delegate (String in) {
9        Console.WriteLine("Hello: " + inStr);
10     };
11     md("My String");
12   }
13 }
```

Listing 55: Delegates in C#

The .net framework provides a built-in concept called *delegates* for handling callbacks. The delegate definition in line 2 is conceptually similar to a Java interface containing only a single method. Line 5 creates an instance of the delegate through an anonymous method implementation. At compile time, the C# compiler creates a new class for each delegate. This class is derived from `System.Delegate`. It declares a constructor and an `Invoke()` method that matches the signature of the declared delegate (`void Invoke(String inStr)` in the example). The constructor takes a reference to the enclosing class instance and the pointer to the method to be called when the delegate is invoked (an int). Since there is one class per delegate and not per implementation, this indirection is required. Note that CIL is able to deal with pointers which is used here for internally managing the delegates.
In the bytecode, invoking a delegate is then represented by simply creating an instance of the delegate class and then calling the `Invoke()` method as shown in Listing 56. At compile time, the anonymous inner method is converted into a normal private method with compiler-generated name. In the example, this is `<test>b__0`. Therefore, no special support for anonymous inner methods is required in the frontend. The opcode `ldftn` is responsible for loading the function pointer of this compiler-generated method onto the stack before invoking the constructor of the delegate class. Note that that the actual bytecode is slightly more complex as the instance of the delegate object is not created anew every time, but cached in a compiler-generated field of the `TestClass` class.

```
1  ldnull
2  ldftn void TestClass::'<test>b__0'(string)
3  newobj instance void TestClass/MyDelegate::.ctor(object, native int)
4  ldstr "My String"
5  callvirt instance void TestClass/MyDelegate::Invoke(string)
```

Listing 56: Simplified Bytecode for Listing 55

The generated delegate class is special, though. Delegates are a concept that is native to the CIL and the .net framework. Therefore, the generated class does not contain any real implementation for the `Invoke()` method or the constructor. Instead, the method is declared as *runtime managed*, a special flag in the method's metadata. This flag instructs the runtime environment to not actually call the empty method when it is invoked. Instead, it jumps to the function pointer that was passed in via the constructor and that is now stored in a field of the delegate class. The process of finding the correct method and invoking it is performed inside the CIL runtime, invisible from the program's code.
To allow existing callgraph algorithms to create sound (and ideally precise) callgraphs, our frontend must emulate this behavior in Jimple code. As function pointers (`ldftn` opcode) do not exist in Jimple, it instead creates an artificial *dispatch* class per function pointer. The `ldftn` opcode is then interpreted as creating an instance of the respective dispatch class and pushing it onto the stack. If the target function is an instance function, the `ldftn` opcode also contains a reference to a target object. This target object is stored in a field of the dispatch class. All these artificial dispatch classes implement a common interface `_cil_delegate` that defines an `Invoke()` method. Since the dispatch class is specific to one single function pointer, it can divert a call to its generic `Invoke()` method to the original method that was referenced in the `ldftn` instruction. This allows the dispatch class to completely cover the semantics of the original function pointer. The `System.Delegate` class must then (instead of the `native int` function pointer) store a reference to a `_cil_delegate` object, i.e., the common interface of all dispatch classes. When the `Invoke()` method of `System.Delegate` is called, it can just call `Invoke` on its artificial dispatch class which in turn calls the target method. Note that the concept of dispatch classes allows the frontend to uniformly handle function pointers in a similar fashion as other load instructions. This works even if the function pointer is not directly used afterwards, but remains on the stack for a while.
Delegates can also be used in asynchronous callbacks. In this case, the caller wants to invoke a delegate and continue with its own execution before the delegate has finished its work. Therefore, generated delegate classes provide two additional methods: `BeginInvoke` and `EndInvoke`. Instead of calling the synchronous `Invoke()` method, client code can also call `BeginInvoke()`. This method takes as an optional parameter a second callback that gets invoked upon completion. Afterwards, the client code can obtain the result of the computation through a call to `EndInvoke()`. In the CIL bytecode, these two additional methods become part of the generated delegate class and thus are easily modeled in Jimple.
A delegate can not only be used to provide a callback to a single method, but also provides multicast support. In the example in Listing 55, a second implementation is added in line 8. When the delegate is invoked, both implementations are called. If a delegate returns a value, the default multicast operations return the value computed by the last invoked implementation. Multicast is, just like unicast, handled by the runtime. The generated delegate class is no different to unicast except for it being derived from `System.MulticastDelegate` instead. To model adding another recipient to a delegate, the compiler first creates a second instance of the delegate class. It then issues a call to the static `Combine()` method the `System.Delegate` API class. This method takes both instances of the delegate class and returns a combined instance. Again, the exact function pointer handling happens inside the CIL runtime and not in user code. In the frontend, we model multicast by chaining dispatch methods. We provide artificial implementations of system methods such as `Delegate.Combine()`. The `Combine()` method, for instance, takes two dispatch classes, and creates a new instance of the first one. This first dispatcher has a reference to the second one which is called in `Invoke()` after the local target (i.e., the first dispatcher's target method) has returned.
Furthermore, note that delegates are objects and can thus freely be passed around in the program. It is legal to create a delegate instance of a private method and then pass this delegate to some other code location from which the target method would not be accessible otherwise. The frontend solves this issue by making all methods public that are referenced through function pointers.

9.3.5 Exceptions

The CIL exception model is very similar to the one of Java. The set of exception that can be thrown by individual statements or expressions, however, is different. In Soot, this was modeled by creating a new implementation of the `ThrowAnalysis` interface to complement the existing implementations for Java and Android. In general, in comparison to Java, CIL opcodes can throw a larger variety of semantically rich exceptions. If an arithmetic operation leads to a numeric overflow, for instance, an `OverflowException` is thrown. A division by zero leads to a `DivideByZeroException` instead of Java's generic `ArithmeticException`. Conceptually, however, the exception analysis is not more complicated.

9.3.6 Reflection

Similar to class constants in Java bytecode, CIL has data structures for reflectively accessing not only classes, but also methods and fields. All of these handles can be loaded using the `ldtoken` opcode. Depending on the argument of the `ldtoken` opcode, an instance of a particular handle class is created. For a class reference, an instance of the `System.Reflection.RuntimeTypeHandle` class is created and put on the stack. Afterwards, one can use the reflection methods in the system class library to perform operations on the handle such as calling a method or accessing a field.
The CIL frontend detects the type of token being loaded. It constructs one data structure per target that is derived from the respective system data structure. If the CIL code creates a reference to class `A`, the frontend generates an artificial class `_cil_typeref_A` that is derived from `System.Reflection.RuntimeTypeHandle`. The class reference is then modeled through an instance of this artificial reference class instead of an instance of the parent system class. In other words, the `ldtoken` opcode is modeled as creating an instance of the artificial handle class. This technique allows the frontend to keep the semantics of the original target without abstracting all references together in a single class. A Jimple class constant would not correctly capture the semantics of class references being structs with methods and fields in CIL. Furthermore, there are no field or method constants in Jimple which would lead to a non-uniform handling of the three token types in CIL.

9.4 Implementation

An assembly containing CIL code is a Windows DLL or EXE file with a proper PE header. These files contain the CIL code as additional resources. The EXE files compiled with Microsoft's compilers also contain small bootstrappers written in native code. This code is responsible for invoking the CIL runtime on the contained managed CIL code without additional effort from the user. If no runtime is installed, it offers to download an install it. Consequently, an assembly is a binary file with complex data structures. To avoid having to parse these binary data structures, our front-end uses `ildasm`, the IL disassembler tool shipped with the Microsoft .net framework. The ildasm tool first converts the binary assembly file into a textual disassembly which Soot's front-end then parses and converts into Jimple code.
We implemented our own parser for CIL disassembly files. Soot is implemented in Java and there is no parser for CIL written in Java yet. Existing work on decompiling CIL assemblies has been performed in CIL by the use of the platform's reflection and code-model APIs. The commercial product *.net Reflector* by Redgate [43], for instance, is implemented as a .net application for this reason.

9.5 Limitations

Some of the .net language features cannot easily be modeled in Jimple. The .net framework, for instance, allows for *mixed-mode DLLs*. Such DLLs are .net assemblies containing CIL code as well as native, platform-specific Windows DLL files. Both parts contain user code. Methods implemented in CIL code can call native methods and vice versa. Native code can construct and use classes in CIL, and various techniques exist for marshaling data transferred between native and CIL code. In contrast to Java's JNI, a mixed-mode DLL in .net integrates native and managed code more tightly. CIL, for instance, supports bytecode instructions that call native methods given their offset in the file. Mixed-mode DLLs are usually written in an extended version of C++ that supports additional modifiers. For the developer, the difference between managed and native code is a matter of adding or leaving away these modifiers. As Jimple is based on Java, the frontend would need to model this tight coupling as explicit calls to native methods which is non-trivial. Therefore, we leave modeling mixed-mode DLLs to future work.

[43] http://www.red-gate.com/products/dotnet-development/reflector/

9.6 Evaluation

In this section, we evaluate the performance of the CIL frontend presented in this paper. We furthermore use the frontend to apply existing analyses to CIL bytecode. We also report on experiments on a recent malware sample for Android that uses CIL code to hide its malicious behavior from state-of-the-art detection tools. It exploits that most of these tools do not support CIL code, although CIL code can be run on Android using the Mono framework.

9.6.1 Performance

A new frontend to any static analysis framework should be able to efficiently handle even large input files. Note that the implementation of the frontend is not yet fully stable and functional for every corner case, which is why the performance data reported here is preliminary. As we have not yet spent any explicit effort on performance optimization, it can be seen as an upper bound for the computation time.
When parsing a simple "Hello World" program written in Java using the ASM-based Java bytecode frontend, Soot loads 216 system classes this program depends on. When loading the semantically equivalent program written in C# using the CIL frontend, Soot needs to load only 114 classes, due to the different structure of the .net framework's runtime. These classes are contained in the `mscorlib` system assembly whose binary is about 5 megabytes and whose disassembly is about 55 megabytes in size. In total, for Microsoft .net framework version 4.0.30319 x64 `mscorlib` comprises more than 3,200 types, 28,300 methods, and 14,200 fields.
In the case of Java, the Jimple conversion requires about one second. In the case of CIL, the current frontend requires six seconds. This excludes the additional time required by Microsoft's external ILDASM tool to disassemble the bytecode. From our experience, however, this time is negligible. At the moment, the biggest bottleneck appears to be the parsing of the huge input text file. We plan to improve the performance in future work.

9.6.2 Cross-Platform Cross-Language Malware for Android

Mobile devices are used to process a great amount of sensitive information such as banking or health data. Furthermore, these devices are equipped with a broad variety of sensors such as for location (GPS) or acceleration. They can also impose charges at the cost of the user by sending SMS messages to costly premium-rate telephone numbers. Unsurprisingly, these features have attracted miscreants who develop and provide malicious apps. Due to the great market share of Android (more than 80%), most mobile malware is developed for Android. Regardless of the programming language used to develop an app, it must be compiled to Dalvik bytecode. Dalvik is a register-based bytecode language that is specially optimized for resource-constrained mobile devices.
Since there is no compiler from .net languages such as C# to Dalvik, any Android application that wishes to use .net must have its CIL code intepreted, through a special version of the Mono framework (an open-source implementation of a CIL runtime) compiled for ARM. The Mono execution environment runs side-by-side with Android's normal Dalvik runtime. A recent malware sample (identifiable through its package name `com.tinker.gameone`) uses CIL to hide its malicious code from purely Dalvik-based analyses, as they are commonly used by mobile app stores to block malware from the store. In the `gameone` malware, these analysis only see the Dalvik bytecode of the benign Mono framework, which is why the app made it into several stores. According to *virustotal.com*, even at the time of writing the paper, the malware was only detected by 29 out of 56 popular anti-virus tools. Those tools use signature-based matching, i.e., can only re-identify the malware once its malicious behavior has been manually identified.
The malware protects its assemblies from being disassembled by associating the `SuppressIldasmAttribute` with its assembly. Recall that in CIL assemblies are proper entities and may thus have attributes associated. This particular attribute is checked by `ildasm`. If present, `ildasm` refuses to disassemble the assembly. We countered this protection by removing the attribute before disassembling the file. In the Jimple code, the class `FBAccount.TinkerAccount` contains a method `AccountSend Data()`. This method calls a number of methods with obfuscated names, which are, however, only wrappers around API calls. In the end, the code calls the method `PushToServer()` in the class `AppData.ClientDataManager`. With our frontend, Soot was able to find call sites for methods such as `getResult` in `System.Net. Http.Http ResponseMessage` as they appear in `PushToServer()`. In total, these methods send the user's Facebook credentials over the internet. Although not yet tested, we are confident that existing data-flow analyses or slicing techniques can be applied as well without modifications to the analysis as the data flow is fairly trivial in this malware app. It furthermore allows the human analyst to read convenient Jimple code instead of the stack-based CIL disassembly. Therefore, using our frontend it would have been easy to detect this malware with existing analysis techniques.

9.7 Related Work

The CIL bytecode language is defined as a part of the Common Language Infrastructure (CLI) which is defined in standard ECMA-335 [32]. More precisely, the language parsed by our framework is the textual ILAsm language defined in Part IV of the standard. Others have added documentation on how high-level languages such as C# compile to CIL code [22].
Existing work such as the inline reference monitoring for .net programs proposed by Hamlen, Morrisett, and Schneider [62] is based on Microsoft's ILX SDK. The ILX SDK is capable of reading and writing .net assemblies with the help of OCAML. This toolkit also extends the CIL language with constructs for closures, functions types, thunks, and others [132]. Microsoft develops the Phoenix Compiler[44] as a research platform for code generation, optimization, and program analysis which can handle native PE binaries as well as CIL code. Phoenix uses a single, strongly-typed intermediate representation. Conceptually, this approach is thus closest to the work presented in this paper. Other purpose-built specialized analysis tools include FxCop (similar to FindBugs [15] in the Java world) and StyleCop for detecting bad code style. Kieker.NET [92] is a framework for performing dynamic analysis on .net programs. It re-uses the original Kieker implementation for Java [64] by the means of a custom interoperability layer.

9.8 Conclusions

We have presented a novel frontend for the Soot program analysis framework. The frontend is capable of converting CIL bytecode into Soot's Jimple intermediate representation. We have shown how CIL language constructs can be expressed in Jimple, though Jimple was originally designed for the less expressive Java bytecode language. As future work, we plan to remove the dependency on `ildasm` and implement a conversion directly from the binary CIL datastructures instead of the disassembly text. We are currently working on integrating our frontend into the Soot open-source framework. As the ultimate goal, we hope to apply a mainly unmodified version of the FlowDroid static data flow tracker to .net programs. The work presented in this section serves as a first step into the direction of building a unified data flow solver for multiple platforms based on FlowDroid.

[44] `http://research.microsoft.com/en-us/collaboration/focus/cs/phoenix.aspx`

10 Conclusion

In this thesis, we have investigated how static data flow analysis can be used to detect privacy leaks in Android apps. Many apps have the necessary privileges to obtain highly-sensitive information from the Android operating system such as the user's unique identifiers (IMEI, IMSI, telephone number, network MAC address, etc.), his personal files, and database entries (address book, calendar, etc.). Furthermore, many apps are granted access to sensor data such as the user's GPS location. Therefore, the user must have a means to ascertain that the app only uses this data in the expected way, and does not leak it to unauthorized third parties. Our statement in this thesis was that static data flow analysis is an adequate technique for checking how apps deal with sensitive information and can serve as an additional input to the user's informed decision of whether he wants to use the app in question (and thereby entrust it with his personal data) or not.

We have presented the FlowDroid static taint tracker as a highly precise and efficient approach for analyzing large real-world Android apps taken from the official Google Play Store. With the DroidBench micro-benchmark suite, we provide testing examples with a known ground truth to assess the performance and precision of the tool. We showed that FlowDroid outperformed the existing tools in both categories. We then applied FlowDroid to real-world apps to assess its performance. We showed that conducting a data flow analysis with FlowDroid is feasible even for large and complex apps. Therefore, we can conclude that with FlowDroid, a user can check real-world apps for privacy leaks before he installs them on his phone. The FlowDroid output helps him regain the sovereignty on his data beyond merely trusting the claims made by the developer or vendor. At the moment, we are working on further improving the performance and stability of FlowDroid in a commercial project with an industry partner. This work will ensure that FlowDroid meets the requirements for large-scale productive use in an industrial setting where the tool will be used for checking Android apps against privacy policies.

We acknowledge that a determined adversary will always find ways to make program behavior in general, or data flows in particular, unavailable to any given static analysis tool. The only way to prevent such attacks is to report a potential privacy violation whenever an app makes use of a language feature the analyzer can not precisely reason about, and advise the user to refrain from installing the respective app. This approach is, however, infeasible in practice, as it would reject many popular benign apps, which hinders user acceptance. If an analysis raises an alarm for too many apps that do not actually infringe upon the user's privacy, the user is likely to ignore the warnings issued by the tool even for those apps for which they are relevant. Therefore, the FlowDroid analysis does not aim to be sound, but complete and precise enough for practical use. Consequently, the attacker can, for instance, place all of the malicious behavior in native code or in dynamically-loaded code files that are only downloaded from a malicious web server right before execution. Nevertheless, we still think FlowDroid is a useful addition to an app analyst's toolbox. Most apps uploaded to stores such as Google Play are not outright malicious. Still, their developers may have different privacy policies than the user or might simply not care as much about privacy as the user does. In such a case, the data flows are not heavily obfuscated, and a tool such as FlowDroid gives useful insights. Additionally, complementary approaches for undoing certain obfuscation techniques such as using reflection instead of normal method calls exist [85, 87, 112, 138]. Such approaches can be run as pre-analysis steps that simply the app before FlowDroid conducts its data flow analysis.

The general concepts we have presented in this thesis are not limited to the Android platform or to Java-based programs in general. We have designed FlowDroid to work for any analysis target whose code can be compiled to the Jimple intermediate representation. Additional semantics that differ between the new platform and what we provide for Android and Java can be modeled by implementing FlowDroid's open interfaces. As a first step into this direction, we presented a front-end that converts code for the Microsoft.net framework to Jimple. In general, we propose FlowDroid as an open-source framework for further research and experimentation in the area of static data flow analysis. We are happy to see that many research groups worldwide have already built their work on the tool, have compared their own tools to FlowDroid, or have used DroidBench in their evaluations. It is our belief that making tools and code available to the research community is essential for independent verification of claims, for building upon existing tools, and for improving the state of the art.

We conclude this thesis in the hope that it provides useful insights on how the mobile users' privacy can be protected better while still being able to enjoy the comfort of large app stores that provide apps for almost any need. We believe that it is important to analyze the existing apps available in the stores today as they are without forcing the app developers to specifically cater for the requirements of the analysis tool. Certification approaches, while they have the ability to provide much greater levels of security, have a much higher adoption barrier when they require the app developers to change their code and fail on legacy code that cannot be adopted. An approach such

as ours might miss a leak, but it has been proven to be useful to detect the important leaks in the important (i.e., widely-used and popular) apps that users download from the stores today.

Own Papers

[4] S. Arzt and E. Bodden. “Reviser: efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes”. In: *Proceedings of the 36th International Conference on Software Engineering*. ACM. 2014, pp. 288–298.

[5] S. Arzt and E. Bodden. “StubDroid: automatic inference of precise data-flow summaries for the android framework”. In: *Proceedings of the 38th International Conference on Software Engineering*. ACM. 2016, pp. 725–735 (cit. on pp. 16, 101).

[6] S. Arzt, T. Kussmaul, and E. Bodden. “Towards cross-platform cross-language analysis with soot”. In: *Proceedings of the 5th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*. ACM. 2016, pp. 1–6 (cit. on p. 153).

[7] S. Arzt, S. Rasthofer, and E. Bodden. “Instrumenting android and java applications as easy as abc”. In: *Runtime Verification*. Springer Berlin Heidelberg, 2013, pp. 364–381.

[8] S. Arzt et al. “Denial-of-App Attack: Inhibiting the Installation of Android Apps on Stock Phones”. In: *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*. ACM. 2014, pp. 21–26.

[9] S. Arzt et al. “Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps”. In: *ACM SIGPLAN Notices*. Vol. 49. 6. ACM. 2014, pp. 259–269 (cit. on pp. 13, 15, 28, 52, 103, 104, 114, 129).

[10] S. Arzt et al. “How useful are existing monitoring languages for securing Android apps?” In: *Software Engineering (Workshops)*. 2013, pp. 107–122.

[11] S. Arzt et al. “Towards secure integration of cryptographic software”. In: *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* ACM. 2015, pp. 1–13.

[12] S. Arzt et al. “Using targeted symbolic execution for reducing false-positives in dataflow analysis”. In: *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*. ACM. 2015, pp. 1–6 (cit. on p. 70).

[86] L. Li et al. “IccTA: detecting inter-component privacy leaks in android apps”. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE 2015)*. 2015 (cit. on pp. 99, 129).

[109] S. Rasthofer, S. Arzt, and E. Bodden. “A machine-learning approach for classifying and categorizing android sources and sinks”. In: *2014 Network and Distributed System Security Symposium (NDSS)* (2014) (cit. on pp. 19, 33, 38).

[110] S. Rasthofer et al. “Droidforce: enforcing complex, data-centric, system-wide policies in android”. In: *Availability, Reliability and Security (ARES), 2014 Ninth International Conference on*. IEEE. 2014, pp. 40–49 (cit. on p. 100).

[111] S. Rasthofer et al. “Droidsearch: A tool for scaling android app triage to real-world app stores”. In: *Proceedings of the IEEE Technically Co-Sponsored Science and Information Conference* (2015) (cit. on p. 62).

[112] S. Rasthofer et al. “Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques”. In: *2016 Network and Distributed System Security Symposium (NDSS)* (2016) (cit. on pp. 97, 128, 162).

[113] S. Rasthofer et al. “(In)Security of Backend-as-a-Service”. In: *Black Hat Europe 2015 Briefing White Papers* (2015) (cit. on p. 10).

References

[1] Y. Aafer, W. Du, and H. Yin. "DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android". In: *Security and Privacy in Communication Networks: 9th International ICST Conference, SecureComm 2013, Sydney, NSW, Australia, September 25-28, 2013, Revised Selected Papers*. Ed. by T. Zia et al. Cham: Springer International Publishing, 2013, pp. 86–103. ISBN: 978-3-319-04283-1. DOI: 10.1007/978-3-319-04283-1_6. URL: http://dx.doi.org/10.1007/978-3-319-04283-1_6 (cit. on p. 153).

[2] G. Agrawal, J. Li, and Q. Su. "Evaluating a Demand Driven Technique for Call Graph Construction". In: *Compiler Construction: 11th International Conference, CC 2002 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8–12, 2002 Proceedings*. Ed. by R. N. Horspool. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 29–45. ISBN: 978-3-540-45937-8. DOI: 10.1007/3-540-45937-5_5. URL: http://dx.doi.org/10.1007/3-540-45937-5_5 (cit. on p. 35).

[3] Android Developer Documentation. *System Permissions*. http://developer.android.com/guide/topics/security/permissions.html. Accessed: 2016-11-10 (cit. on p. 12).

[4] S. Arzt and E. Bodden. "Reviser: efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes". In: *Proceedings of the 36th International Conference on Software Engineering*. ACM. 2014, pp. 288–298.

[5] S. Arzt and E. Bodden. "StubDroid: automatic inference of precise data-flow summaries for the android framework". In: *Proceedings of the 38th International Conference on Software Engineering*. ACM. 2016, pp. 725–735 (cit. on pp. 16, 101).

[6] S. Arzt, T. Kussmaul, and E. Bodden. "Towards cross-platform cross-language analysis with soot". In: *Proceedings of the 5th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*. ACM. 2016, pp. 1–6 (cit. on p. 153).

[7] S. Arzt, S. Rasthofer, and E. Bodden. "Instrumenting android and java applications as easy as abc". In: *Runtime Verification*. Springer Berlin Heidelberg, 2013, pp. 364–381.

[8] S. Arzt et al. "Denial-of-App Attack: Inhibiting the Installation of Android Apps on Stock Phones". In: *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*. ACM. 2014, pp. 21–26.

[9] S. Arzt et al. "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps". In: *ACM SIGPLAN Notices*. Vol. 49. 6. ACM. 2014, pp. 259–269 (cit. on pp. 13, 15, 28, 52, 103, 104, 114, 129).

[10] S. Arzt et al. "How useful are existing monitoring languages for securing Android apps?" In: *Software Engineering (Workshops)*. 2013, pp. 107–122.

[11] S. Arzt et al. "Towards secure integration of cryptographic software". In: *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* ACM. 2015, pp. 1–13.

[12] S. Arzt et al. "Using targeted symbolic execution for reducing false-positives in dataflow analysis". In: *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*. ACM. 2015, pp. 1–6 (cit. on p. 70).

[13] K. W. Y. Au et al. "Pscout: analyzing the android permission specification". In: *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM. 2012, pp. 217–228 (cit. on p. 19).

[14] V. Avdiienko et al. "Mining apps for abnormal usage of sensitive data". In: *2015 International Conference on Software Engineering (ICSE)*. 2015 (cit. on p. 98).

[15] N. Ayewah et al. "Using Static Analysis to Find Bugs". In: *IEEE Software* 25.5 (2008), pp. 22–29. ISSN: 0740-7459. DOI: 10.1109/MS.2008.130 (cit. on p. 161).

[16] M. Backes et al. "AppGuard - Enforcing User Requirements on Android Apps". English. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by N. Piterman and S. Smolka. Vol. 7795. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 543–548. ISBN: 978-3-642-36741-0. DOI: 10.1007/978-3-642-36742-7_39. URL: http://dx.doi.org/10.1007/978-3-642-36742-7_39 (cit. on p. 11).

[17] G. Barbon et al. “Privacy Analysis of Android Apps: Implicit Flows and Quantitative Analysis”. In: *Computer Information Systems and Industrial Management: 14th IFIP TC 8 International Conference, CISIM 2015, Warsaw, Poland, September 24-26, 2015, Proceedings*. Ed. by K. Saeed and W. Homenda. Cham: Springer International Publishing, 2015, pp. 3–23. ISBN: 978-3-319-24369-6. DOI: 10.1007/978-3-319-24369-6_1. URL: http://dx.doi.org/10.1007/978-3-319-24369-6_1 (cit. on p. 98).

[18] A. Bartel et al. “Dexpler: converting Android Dalvik bytecode to Jimple for static analysis with Soot”. In: *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*. SOAP ’12. 2012, pp. 27–38 (cit. on pp. 29, 153).

[19] O. Bastani, S. Anand, and A. Aiken. “Interactively Verifying Absence of Explicit Information Flows in Android Apps”. In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2015. Pittsburgh, PA, USA: ACM, 2015, pp. 299–315. ISBN: 978-1-4503-3689-5. DOI: 10.1145/2814270.2814274. URL: http://doi.acm.org/10.1145/2814270.2814274 (cit. on p. 97).

[20] A. R. Beresford et al. “MockDroid: Trading Privacy for Application Functionality on Smartphones”. In: *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*. HotMobile ’11. Phoenix, Arizona: ACM, 2011, pp. 49–54. ISBN: 978-1-4503-0649-2. DOI: 10.1145/2184489.2184500. URL: http://doi.acm.org/10.1145/2184489.2184500 (cit. on p. 12).

[21] S. Blackshear, A. Gendreau, and B.-Y. E. Chang. “Droidel: A General Approach to Android Framework Modeling”. In: *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*. SOAP 2015. Portland, OR, USA: ACM, 2015, pp. 19–25. ISBN: 978-1-4503-3585-0. DOI: 10.1145/2771284.2771288. URL: http://doi.acm.org/10.1145/2771284.2771288 (cit. on p. 96).

[22] J. Bock. *CIL Programming: Under the Hood of. NET*. Apress, 2008 (cit. on p. 161).

[23] E. Bodden. “Inter-procedural data-flow analysis with IFDS/IDE and Soot”. In: *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*. SOAP ’12. 2012, pp. 3–8 (cit. on pp. 25, 65, 149).

[24] M. Bravenboer and Y. Smaragdakis. “Strictly Declarative Specification of Sophisticated Points-to Analyses”. In: *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA ’09. Orlando, Florida, USA: ACM, 2009, pp. 243–262. ISBN: 978-1-60558-766-0. DOI: 10.1145/1640089.1640108. URL: http://doi.acm.org/10.1145/1640089.1640108 (cit. on p. 57).

[25] J. Breitner et al. “On Improvements of Low-Deterministic Security”. In: *Principles of Security and Trust: 5th International Conference, POST 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Ed. by F. Piessens and L. Viganò. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 68–88. ISBN: 978-3-662-49635-0. DOI: 10.1007/978-3-662-49635-0_4. URL: http://dx.doi.org/10.1007/978-3-662-49635-0_4 (cit. on p. 82).

[26] Business Insider. *Facebook is officially a mobile-first company*. http://www.businessinsider.com/facebook-mobile-only-users-most-common-2015-11. Accessed: 2016-11-10 (cit. on p. 9).

[27] D. Callahan et al. “Interprocedural Constant Propagation”. In: *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*. SIGPLAN ’86. Palo Alto, California, USA: ACM, 1986, pp. 152–161. ISBN: 0-89791-197-0. DOI: 10.1145/12276.13327. URL: http://doi.acm.org/10.1145/12276.13327 (cit. on p. 74).

[28] S. Calzavara, I. Grishchenko, and M. Maffei. “HornDroid: Practical and Sound Static Analysis of Android Applications by SMT Solving”. In: *2016 IEEE European Symposium on Security and Privacy (EuroS P)*. 2016, pp. 47–62. DOI: 10.1109/EuroSP.2016.16 (cit. on p. 98).

[29] Y. Cao et al. “EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework.” In: *2015 Network and Distributed System Security Symposium (NDSS)*. 2015 (cit. on p. 96).

[30] K. Z. Chen et al. “Contextual Policy Enforcement in Android Applications with Permission Event Graphs.” In: 2013 (cit. on p. 12).

[31] S. R. Choudhary, A. Gorla, and A. Orso. “Automated Test Input Generation for Android: Are We There Yet?(E)”. In: *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE. 2015, pp. 429–440 (cit. on p. 14).

[32] *Common Language Infrastructure (CLI)*. Norm. 2012 (cit. on p. 161).

[33] K. Coogan et al. “Automatic Static Unpacking of Malware Binaries”. In: *WCRE ’09*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 167–176. ISBN: 978-0-7695-3867-9. DOI: 10.1109/WCRE.2009.24 (cit. on p. 87).

[34] A. De and D. D’Souza. “Scalable Flow-sensitive Pointer Analysis for Java with Strong Updates”. In: *Proceedings of the 26th European Conference on Object-Oriented Programming*. ECOOP’12. Beijing, China: Springer-Verlag, 2012, pp. 665–687. ISBN: 978-3-642-31056-0. DOI: 10.1007/978-3-642-31057-7_29. URL: http://dx.doi.org/10.1007/978-3-642-31057-7_29 (cit. on p. 58).

[35] A. De and D. DâĂŹSouza. “Scalable Flow-Sensitive Pointer Analysis for Java with Strong Updates”. English. In: *ECOOP 2012 âĂŞ Object-Oriented Programming*. Ed. by J. Noble. Vol. 7313. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 665–687. ISBN: 978-3-642-31056-0. DOI: 10.1007/978-3-642-31057-7_29. URL: http://dx.doi.org/10.1007/978-3-642-31057-7_29 (cit. on p. 21).

[36] A. Desnos et al. “Androguard-Reverse engineering, Malware and goodware analysis of Android applications”. In: *URL code. google. com/p/androguard* (2013) (cit. on p. 153).

[37] A. Deutsch. “Interprocedural May-alias Analysis for Pointers: Beyond K-limiting”. In: *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*. PLDI ’94. Orlando, Florida, USA: ACM, 1994, pp. 230–241. ISBN: 0-89791-662-X. DOI: 10.1145/178243.178263. URL: http://doi.acm.org/10.1145/178243.178263 (cit. on pp. 22, 23).

[38] I. Dillig, T. Dillig, and A. Aiken. “Precise reasoning for programs using containers”. In: *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL ’11. 2011, pp. 187–200 (cit. on p. 40).

[39] Q. Do, B. Martini, and K.-K. Choo. “Enhancing User Privacy on Android Mobile Devices via Permissions Removal”. In: *System Sciences (HICSS), 2014 47th Hawaii International Conference on*. 2014, pp. 5070–5079. DOI: 10.1109/HICSS.2014.623 (cit. on p. 11).

[40] W. Enck, M. Ongtang, and P. McDaniel. “On lightweight mobile phone application certification”. In: *Proceedings of the 16th ACM conference on Computer and communications security*. ACM. 2009, pp. 235–245 (cit. on p. 12).

[41] W. Enck et al. “TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones”. In: *OSDI*. 2010, pp. 393–407 (cit. on p. 14).

[42] S. Ereth and H. Mantel. *Towards a Common Specification Language for Information-Flow Security in RS 3 and Beyond: RIFL 1.0–The Language*. Tech. rep. TUD-CS-2014-0115. MAIS, Computer Science, TU Darmstadt, 2015. URL: http://www.mais.informatik.tu-darmstadt.de/WebBibPHP/papers/2014/RIFL1.0-TechnicalReport-Revision1.pdf (cit. on p. 20).

[43] M. D. Ernst et al. “Collaborative Verification of Information Flow for a High-Assurance App Store”. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’14. Scottsdale, Arizona, USA: ACM, 2014, pp. 1092–1104. ISBN: 978-1-4503-2957-6. DOI: 10.1145/2660267.2660343. URL: http://doi.acm.org/10.1145/2660267.2660343 (cit. on p. 97).

[44] Facebook Advertiser Help Center. *Audience Targeting Options*. https://www.facebook.com/business/help/633474486707199. Accessed: 2016-11-10 (cit. on p. 10).

[45] Facebook Business. *How to target Facebook Ads*. https://www.facebook.com/business/a/online-sales/ad-targeting-details. Accessed: 2016-11-10 (cit. on p. 10).

[46] S. Fahl et al. “Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security”. In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS ’12. Raleigh, North Carolina, USA: ACM, 2012, pp. 50–61. ISBN: 978-1-4503-1651-4. DOI: 10.1145/2382196.2382205. URL: http://doi.acm.org/10.1145/2382196.2382205 (cit. on p. 153).

[47] A. P. Felt et al. “Android Permissions Demystified”. In: *Proceedings of the 18th ACM Conference on Computer and Communications Security*. CCS ’11. Chicago, Illinois, USA: ACM, 2011, pp. 627–638. ISBN: 978-1-4503-0948-6. DOI: 10.1145/2046707.2046779. URL: http://doi.acm.org/10.1145/2046707.2046779 (cit. on p. 19).

[48] A. P. Felt et al. “Android permissions: User attention, comprehension, and behavior”. In: *Proceedings of the Eighth Symposium on Usable Privacy and Security*. ACM. 2012, p. 3 (cit. on p. 11).

[49] Y. Feng et al. *Apposcopy: Semantics-Based Detection of Android Malware*. Tech. rep. submitted for publication. Stanford University, 2013 (cit. on pp. 101, 114).

[50] J. Ferrante, K. J. Ottenstein, and J. D. Warren. "The program dependence graph and its use in optimization". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9.3 (1987), pp. 319–349 (cit. on p. 14).

[51] P. Ferrara, O. Tripp, and M. Pistoia. "MorphDroid: Fine-grained Privacy Verification". In: *Proceedings of the 31st Annual Computer Security Applications Conference*. ACSAC 2015. Los Angeles, CA, USA: ACM, 2015, pp. 371–380. ISBN: 978-1-4503-3682-6. DOI: 10.1145/2818000.2818037. URL: http://doi.acm.org/10.1145/2818000.2818037 (cit. on p. 98).

[52] Forbes Entrepreneurs. *How to Build a Mobile-First Company*. http://www.forbes.com/sites/tomtaulli/2013/03/21/how-to-build-a-mobile-first-company/. Accessed: 2016-11-10 (cit. on p. 9).

[53] C. Fritz. "Flowdroid: A precise and scalable data flow analysis for android". MA thesis. 2013 (cit. on p. 16).

[54] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. "Scandroid: Automated security certification of android applications". In: *Manuscript, Univ. of Maryland, http://www. cs. umd. edu/avik/projects/scandroidascaa* 2.3 (2009) (cit. on pp. 13, 101).

[55] C. Gibler et al. "AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale". English. In: *Trust and Trustworthy Computing*. Ed. by S. Katzenbeisser et al. Vol. 7344. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 291–307. ISBN: 978-3-642-30920-5. DOI: 10.1007/978-3-642-30921-2_17. URL: http://dx.doi.org/10.1007/978-3-642-30921-2_17 (cit. on pp. 13, 19).

[56] D. Giffhorn and C. Hammer. "Precise analysis of java programs using joana". In: *Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference on*. IEEE. 2008, pp. 267–268 (cit. on p. 14).

[57] Google AdWords Help. *About demographic targeting*. https://support.google.com/adwords/answer/2580383?hl=en. Accessed: 2016-11-10 (cit. on p. 10).

[58] Google AdWords Help. *Target ads to geographic locations*. https://support.google.com/adwords/answer/1722043?hl=en. Accessed: 2016-11-10 (cit. on p. 9).

[59] Google Inside AdWords. *Building for the next moment*. https://adwords.googleblog.com/2015/05/building-for-next-moment.html. Accessed: 2016-11-10 (cit. on p. 9).

[60] M. I. Gordon et al. "Information-flow analysis of Android applications in DroidSafe". In: *Proc. of the Network and Distributed System Security Symposium (NDSS). The Internet Society*. 2015 (cit. on pp. 14, 19, 62, 96, 101, 103, 114, 119, 153).

[61] J. Graf, M. Hecker, and M. Mohr. "Using JOANA for Information Flow Control in Java Programs-A Practical Guide." In: *Software Engineering (Workshops)*. 2013, pp. 123–138 (cit. on p. 14).

[62] K. W. Hamlen, G. Morrisett, and F. B. Schneider. "Certified In-lined Reference Monitoring on .NET". In: *Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security*. PLAS '06. Ottawa, Ontario, Canada: ACM, 2006, pp. 7–16. ISBN: 1-59593-374-3. DOI: 10.1145/1134744.1134748. URL: http://doi.acm.org/10.1145/1134744.1134748 (cit. on p. 161).

[63] N. Heintze and O. Tardieu. "Demand-driven Pointer Analysis". In: *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*. PLDI '01. Snowbird, Utah, USA: ACM, 2001, pp. 24–34. ISBN: 1-58113-414-2. DOI: 10.1145/378795.378802. URL: http://doi.acm.org/10.1145/378795.378802 (cit. on p. 58).

[64] A. van Hoorn et al. *Continuous Monitoring of Software Services: Design and Application of the Kieker Framework*. Research Report. Kiel University, 2009. URL: http://oceanrep.geomar.de/14459/ (cit. on p. 161).

[65] P. Hornyack et al. "These Aren'T the Droids You'Re Looking for: Retrofitting Android to Protect Data from Imperious Applications". In: *Proceedings of the 18th ACM Conference on Computer and Communications Security*. CCS '11. Chicago, Illinois, USA: ACM, 2011, pp. 639–652. ISBN: 978-1-4503-0948-6. DOI: 10.1145/2046707.2046780. URL: http://doi.acm.org/10.1145/2046707.2046780 (cit. on p. 12).

[66] J. Huang et al. “SUPOR: Precise and Scalable Sensitive User Input Detection for Android Apps”. In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 977–992. ISBN: 978-1-931971-232. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/huang (cit. on p. 96).

[67] W. Huang, Y. D. A. Milanova, and J. Dolby. *Scalable and Precise Taint Analysis for Android*. Tech. rep. Technical report, Department of Computer Science, Rensselaer Polytechnic Institute, 2015 (cit. on pp. 59, 101).

[68] W. Huang et al. “Scalable and Precise Taint Analysis for Android”. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ISSTA 2015. Baltimore, MD, USA: ACM, 2015, pp. 106–117. ISBN: 978-1-4503-3620-8. DOI: 10.1145/2771783.2771803. URL: http://doi.acm.org/10.1145/2771783.2771803 (cit. on p. 97).

[69] J. Jeon et al. “Dr. Android and Mr. Hide: fine-grained permissions in android applications”. In: *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*. ACM. 2012, pp. 3–14 (cit. on p. 11).

[70] A. Johnson et al. “Exploring and Enforcing Security Guarantees via Program Dependence Graphs”. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’15. Portland, OR, USA: ACM, 2015, pp. 291–302. ISBN: 978-1-4503-3468-6. DOI: 10.1145/2737924.2737957. URL: http://doi.acm.org/10.1145/2737924.2737957 (cit. on p. 98).

[71] N. D. Jones and S. S. Muchnick. “Flow Analysis and Optimization of LISP-like Structures”. In: *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL ’79. San Antonio, Texas: ACM, 1979, pp. 244–256. DOI: 10.1145/567752.567776. URL: http://doi.acm.org/10.1145/567752.567776 (cit. on p. 22).

[72] V. Kanvar and U. P. Khedker. “Heap Abstractions for Static Analysis”. In: *ACM Comput. Surv.* 49.2 (June 2016), 29:1–29:47. ISSN: 0360-0300. DOI: 10.1145/2931098. URL: http://doi.acm.org/10.1145/2931098 (cit. on pp. 24, 48).

[73] P. Kelley et al. “A Conundrum of Permissions: Installing Applications on an Android Smartphone”. English. In: *Financial Cryptography and Data Security*. Ed. by J. Blyth, S. Dietrich, and L. Camp. Vol. 7398. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 68–79. ISBN: 978-3-642-34637-8. DOI: 10.1007/978-3-642-34638-5_6. URL: http://dx.doi.org/10.1007/978-3-642-34638-5_6 (cit. on p. 11).

[74] U. P. Khedker, A. Sanyal, and A. Karkare. “Heap Reference Analysis Using Access Graphs”. In: *ACM Trans. Program. Lang. Syst.* 30.1 (Nov. 2007). ISSN: 0164-0925. DOI: 10.1145/1290520.1290521. URL: http://doi.acm.org/10.1145/1290520.1290521 (cit. on p. 23).

[75] J. Kim et al. “ScanDal: Static Analyzer for Detecting Privacy Leaks in Android Applications”. In: *MoST 2012: Mobile Security Technologies 2012*. Ed. by H. Chen, L. Koved, and D. S. Wallach. San Francisco, CA, USA: IEEE, May 2012. URL: http://ropas.snu.ac.kr/scandal/ (cit. on pp. 14, 101, 114, 153).

[76] D. King et al. “Information Systems Security: 4th International Conference, ICISS 2008, Hyderabad, India, December 16-20, 2008. Proceedings”. In: ed. by R. Sekar and A. K. Pujari. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. Chap. Implicit Flows: Can’t Live with ‘Em, Can’t Live without ‘Em, pp. 56–70. ISBN: 978-3-540-89862-7. DOI: 10.1007/978-3-540-89862-7_4. URL: http://dx.doi.org/10.1007/978-3-540-89862-7_4 (cit. on pp. 43, 107).

[77] W. Klieber et al. “Android taint flow analysis for app sets”. In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*. ACM. 2014, pp. 1–6 (cit. on pp. 99, 129).

[78] P. Lam et al. “The Soot framework for Java program analysis: a retrospective”. In: *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*. 2011 (cit. on pp. 18, 28, 105, 106).

[79] G. T. Leavens, A. L. Baker, and C. Ruby. “JML: A notation for detailed design”. In: *Behavioral specifications of Businesses and Systems*. Springer, 1999, pp. 175–188 (cit. on p. 104).

[80] J. Lerch and B. Hermann. “Design your analysis: a case study on implementation reusability of data-flow functions”. In: *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*. ACM. 2015, pp. 26–30 (cit. on p. 37).

[81] J. Lerch et al. “Access-Path Abstraction: Scaling Field-Sensitive Data-Flow Analysis With Unbounded Access Paths”. In: () (cit. on p. 23).

[82] J. Lerch et al. “FlowTwist: efficient context-sensitive inside-out taint analysis for large codebases”. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2014, pp. 98–108 (cit. on p. 69).

[83] O. Lhoták and L. Hendren. “Evaluating the Benefits of Context-sensitive Points-to Analysis Using a BDD-based Implementation”. In: *ACM Trans. Softw. Eng. Methodol.* 18.1 (Oct. 2008), 3:1–3:53. ISSN: 1049-331X. DOI: 10.1145/1391984.1391987. URL: http://doi.acm.org/10.1145/1391984.1391987 (cit. on pp. 36, 57).

[84] O. Lhoták and L. Hendren. “Scaling Java Points-to Analysis Using Spark”. English. In: *Compiler Construction*. Ed. by G. Hedin. Vol. 2622. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 153–169. ISBN: 978-3-540-00904-7. DOI: 10.1007/3-540-36579-6_12. URL: http://dx.doi.org/10.1007/3-540-36579-6_12 (cit. on pp. 35, 57, 76).

[85] L. Li et al. “DroidRA: Taming Reflection to Support Whole-program Analysis of Android Apps”. In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ISSTA 2016. Saarbrücken, Germany: ACM, 2016, pp. 318–329. ISBN: 978-1-4503-4390-9. DOI: 10.1145/2931037.2931044. URL: http://doi.acm.org/10.1145/2931037.2931044 (cit. on pp. 96, 162).

[86] L. Li et al. “IccTA: detecting inter-component privacy leaks in android apps”. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE 2015)*. 2015 (cit. on pp. 99, 129).

[87] L. Li et al. “Reflection-aware Static Analysis of Android Apps”. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE 2016. Singapore, Singapore: ACM, 2016, pp. 756–761. ISBN: 978-1-4503-3845-5. DOI: 10.1145/2970276.2970277. URL: http://doi.acm.org/10.1145/2970276.2970277 (cit. on pp. 96, 162).

[88] S. Liang et al. “Sound and precise malware analysis for android via pushdown reachability and entry-point saturation”. In: *Proceedings of the Third ACM workshop on Security and privacy in smartphones & mobile devices*. ACM. 2013, pp. 21–32 (cit. on p. 97).

[89] B. Livshits et al. “Merlin: Specification Inference for Explicit Information Flow Problems”. In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’09. Dublin, Ireland: ACM, 2009, pp. 75–86. ISBN: 978-1-60558-392-1. DOI: 10.1145/1542476.1542485. URL: http://doi.acm.org/10.1145/1542476.1542485 (cit. on p. 19).

[90] S. Lortz et al. “Cassandra: Towards a Certifying App Store for Android”. In: *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*. ACM. 2014, pp. 93–104 (cit. on pp. 97, 101).

[91] L. Lu et al. “CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities”. In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS ’12. Raleigh, North Carolina, USA: ACM, 2012, pp. 229–240. ISBN: 978-1-4503-1651-4. DOI: 10.1145/2382196.2382223. URL: http://doi.acm.org/10.1145/2382196.2382223 (cit. on pp. 13, 101, 103, 109, 114).

[92] F. Magedanz. “Dynamic analysis of .NET applications for architecture-based model extraction and test generation”. Diploma thesis. Kiel University, 2011. URL: http://oceanrep.geomar.de/15486/ (cit. on p. 161).

[93] D. Maier, T. MÃijller, and M. Protsenko. “Divide-and-Conquer: Why Android Malware Cannot Be Stopped”. In: *Availability, Reliability and Security (ARES), 2014 Ninth International Conference on*. 2014, pp. 30–39. DOI: 10.1109/ARES.2014.12 (cit. on p. 14).

[94] H. v. d. Merwe. “Verification of Android Applications”. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 2. 2015, pp. 931–934. DOI: 10.1109/ICSE.2015.295 (cit. on p. 97).

[95] M. Might. “Environment analysis of higher-order languages”. PhD thesis. Georgia Institute of Technology, 2007 (cit. on p. 97).

[96] A. Milanova, W. Huang, and Y. Dong. “CFL-reachability and Context-sensitive Integrity Types”. In: *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. PPPJ ’14. Cracow, Poland: ACM, 2014, pp. 99–109. ISBN: 978-1-4503-2926-2. DOI: 10.1145/2647508.2647522. URL: http://doi.acm.org/10.1145/2647508.2647522 (cit. on p. 97).

[97] M. Mohr, J. Graf, and M. Hecker. “JoDroid: Adding Android Support to a Static Information Flow Control Tool.” In: *Software Engineering (Workshops)*. 2015, pp. 140–145 (cit. on pp. 14, 98).

[98] L. de Moura and N. Bjørner. “Z3: An Efficient SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Ed. by C. R. Ramakrishnan and J. Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24. URL: http://dx.doi.org/10.1007/978-3-540-78800-3_24 (cit. on p. 39).

[99] N. A. Naeem and O. Lhoták. “Faster Alias Set Analysis Using Summaries”. In: *Compiler Construction*. Ed. by J. Knoop. Vol. 6601. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 82–103. ISBN: 978-3-642-19860-1. DOI: 10.1007/978-3-642-19861-8_6. URL: http://dx.doi.org/10.1007/978-3-642-19861-8_6 (cit. on p. 114).

[100] N. A. Naeem, O. Lhoták, and J. Rodriguez. “Practical Extensions to the IFDS Algorithm”. In: *Compiler Construction: 19th International Conference, CC 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. Ed. by R. Gupta. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 124–144. ISBN: 978-3-642-11970-5. DOI: 10.1007/978-3-642-11970-5_8. URL: http://dx.doi.org/10.1007/978-3-642-11970-5_8 (cit. on pp. 25, 27, 37).

[101] Y. Nan et al. “UIPicker: User-Input Privacy Identification in Mobile Applications”. In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 993–1008. ISBN: 978-1-931971-232. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/nan (cit. on p. 96).

[102] D. Octeau et al. “Composite Constant Propagation: Application to Android Inter-component Communication Analysis”. In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. ICSE ’15. Florence, Italy: IEEE Press, 2015, pp. 77–88. ISBN: 978-1-4799-1934-5. URL: http://dl.acm.org/citation.cfm?id=2818754.2818767 (cit. on p. 96).

[103] D. Octeau et al. “Composite constant propagation: Application to android inter-component communication analysis”. In: *Proceedings of the 37th International Conference on Software Engineering (ICSE)*. 2015 (cit. on pp. 99, 129).

[104] D. Octeau et al. “Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis”. In: *USENIX Security 2013*. 2013 (cit. on p. 99).

[105] R. Padhye. “Interprocedural Heap Analysis using Access Graphs and Value Contexts”. PhD thesis. Indian Institute of Technology Bombay, 2013 (cit. on p. 23).

[106] N. J. Percoco and S. Schulte. “Adventures in bouncerland”. In: *Blackhat USA* (2012) (cit. on p. 14).

[107] C. Qian et al. “On tracking information flows through JNI in android applications”. In: *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*. IEEE. 2014, pp. 180–191 (cit. on p. 62).

[108] D. Quinlan. “ROSE: Compiler support for object-oriented frameworks”. In: *Parallel Processing Letters* 10.02n03 (2000), pp. 215–226 (cit. on p. 62).

[109] S. Rasthofer, S. Arzt, and E. Bodden. “A machine-learning approach for classifying and categorizing android sources and sinks”. In: *2014 Network and Distributed System Security Symposium (NDSS)* (2014) (cit. on pp. 19, 33, 38).

[110] S. Rasthofer et al. “Droidforce: enforcing complex, data-centric, system-wide policies in android”. In: *Availability, Reliability and Security (ARES), 2014 Ninth International Conference on*. IEEE. 2014, pp. 40–49 (cit. on p. 100).

[111] S. Rasthofer et al. “Droidsearch: A tool for scaling android app triage to real-world app stores”. In: *Proceedings of the IEEE Technically Co-Sponsored Science and Information Conference* (2015) (cit. on p. 62).

[112] S. Rasthofer et al. “Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques”. In: *2016 Network and Distributed System Security Symposium (NDSS)* (2016) (cit. on pp. 97, 128, 162).

[113] S. Rasthofer et al. “(In)Security of Backend-as-a-Service”. In: *Black Hat Europe 2015 Briefing White Papers* (2015) (cit. on p. 10).

[114] M. Reif et al. “Call graph construction for Java libraries”. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2016, pp. 474–486 (cit. on p. 76).

[115] A. Reina, A. Fattori, and L. Cavallaro. “A System Call-Centric Analysis and Stimulation Technique to Automatically Reconstruct Android Malware Behaviors”. In: *EUROSEC*. Prague, Czech Republic, 2013 (cit. on p. 14).

[116] T. Reps. “Program analysis via graph reachability”. In: *Information and software technology* 40.11 (1998), pp. 701–726 (cit. on p. 97).

[117] T. Reps, S. Horwitz, and M. Sagiv. “Precise interprocedural dataflow analysis via graph reachability”. In: *POPL ’95*. 1995, pp. 49–61 (cit. on pp. 24, 25, 28, 114).

[118] T. Reps et al. “Weighted pushdown systems and their application to interprocedural dataflow analysis”. In: *Science of Computer Programming* 58.1 (2005), pp. 206–263 (cit. on p. 97).

[119] A. Rountev, M. Sharp, and G. Xu. “IDE Dataflow Analysis in the Presence of Large Object-Oriented Libraries”. In: *Compiler Construction*. Ed. by L. Hendren. Vol. 4959. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 53–68. ISBN: 978-3-540-78790-7. DOI: 10.1007/978-3-540-78791-4_4. URL: http://dx.doi.org/10.1007/978-3-540-78791-4_4 (cit. on pp. 108, 114).

[120] M. Sagiv, T. Reps, and S. Horwitz. “Precise interprocedural dataflow analysis with applications to constant propagation”. In: *Theoretical Computer Science* 167.1 (1996), pp. 131–170 (cit. on pp. 65, 114).

[121] L. Shang, X. Xie, and J. Xue. “On-demand Dynamic Summary-based Points-to Analysis”. In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. CGO ’12. San Jose, California: ACM, 2012, pp. 264–274. ISBN: 978-1-4503-1206-6. DOI: 10.1145/2259016.2259050. URL: http://doi.acm.org/10.1145/2259016.2259050 (cit. on p. 58).

[122] S. Shekhar, M. Dietz, and D. S. Wallach. “AdSplit: Separating Smartphone Advertising from Applications”. In: *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX, 2012, pp. 553–567. ISBN: 978-931971-95-9. URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/shekhar (cit. on p. 61).

[123] Smart Insights. *Mobile Marketing Statistics Compilation*. http://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/. Accessed: 2016-11-10 (cit. on p. 9).

[124] F. Song and T. Touili. “Model-Checking for Android Malware Detection”. In: *Programming Languages and Systems: 12th Asian Symposium, APLAS 2014, Singapore, Singapore, November 17-19, 2014, Proceedings*. Ed. by J. Garrigue. Cham: Springer International Publishing, 2014, pp. 216–235. ISBN: 978-3-319-12736-1. DOI: 10.1007/978-3-319-12736-1_12. URL: http://dx.doi.org/10.1007/978-3-319-12736-1_12 (cit. on p. 97).

[125] A. L. Souter and L. L. Pollock. “Incremental Call Graph Reanalysis for Object-Oriented Software Maintenance”. In: *Proceedings of the IEEE International Conference on Software Maintenance (ICSM’01)*. ICSM ’01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 682–. ISBN: 0-7695-1189-9. DOI: 10.1109/ICSM.2001.972787. URL: http://dx.doi.org/10.1109/ICSM.2001.972787 (cit. on p. 90).

[126] J. Späth et al. “Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java”. In: *European Conference on Object-Oriented Programming (ECOOP)*. 2016 (cit. on p. 58).

[127] M. Sridharan et al. “Aliasing in Object-Oriented Programming”. In: ed. by D. Clarke, J. Noble, and T. Wrigstad. Berlin, Heidelberg: Springer-Verlag, 2013. Chap. Alias Analysis for Object-oriented Programs, pp. 196–232. ISBN: 978-3-642-36945-2. URL: http://dl.acm.org/citation.cfm?id=2554511.2554523 (cit. on p. 58).

[128] M. Sridharan et al. “Demand-driven Points-to Analysis for Java”. In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA ’05. San Diego, CA, USA: ACM, 2005, pp. 59–76. ISBN: 1-59593-031-0. DOI: 10.1145/1094811.1094817. URL: http://doi.acm.org/10.1145/1094811.1094817 (cit. on p. 35).

[129] M. Sridharan et al. “Demand-driven Points-to Analysis for Java”. In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA ’05. San Diego, CA, USA: ACM, 2005, pp. 59–76. ISBN: 1-59593-031-0. DOI: 10.1145/1094811.1094817. URL: http://doi.acm.org/10.1145/1094811.1094817 (cit. on p. 36).

[130] M. Sridharan et al. “F4F: Taint Analysis of Framework-based Web Applications”. In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA ’11. Portland, Oregon, USA: ACM, 2011, pp. 1053–1068. ISBN: 978-1-4503-0940-0. DOI: 10.1145/2048066.2048145. URL: http://doi.acm.org/10.1145/2048066.2048145 (cit. on pp. 14, 114).

[131] statista - The Statistics Portal. *Global mobile OS market share in sales to end users from 1st quarter 2009 to 1st quarter 2016*. http://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/. Accessed: 2016-11-10 (cit. on p. 15).

[132] D. Syme. “ILX: Extending the .NET Common {IL} for Functional Language Interoperability”. In: *Electronic Notes in Theoretical Computer Science* 59.1 (2001). BABEL’01, First International Workshop on Multi-Language Infrastructure and Interoperability (Satellite Event of {PLI} 2001), pp. 53 –72. ISSN: 1571-0661. DOI: http://dx.doi.org/10.1016/S1571-0661(05)80453-0 (cit. on p. 161).

[133] The Next Web. *Android users have an average of 95 apps installed on their phones, according to Yahoo Aviate data*. http://thenextweb.com/apps/2014/08/26/android-users-average-95-apps-installed-phones-according-yahoo-aviate-data/. Accessed: 2016-11-10 (cit. on p. 9).

[134] Trend Micro TrendLabs Security Intelligence Blog. *Bypassing Android Permissions: What You Need to Know*. http://blog.trendmicro.com/trendlabs-security-intelligence/bypassing-android-permissions-what-you-need-to-know/. Accessed: 2015-12-23 (cit. on p. 12).

[135] O. Tripp et al. “Andromeda: Accurate and Scalable Security Analysis of Web Applications”. English. In: *Fundamental Approaches to Software Engineering*. Ed. by V. Cortellessa and D. VarrÃş. Vol. 7793. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 210–225. ISBN: 978-3-642-37056-4. DOI: 10.1007/978-3-642-37057-1_15. URL: http://dx.doi.org/10.1007/978-3-642-37057-1_15 (cit. on pp. 21, 58, 104).

[136] N. Umanee. “Shimple: An investigation of static single assignment form”. PhD thesis. McGill University, 2005 (cit. on p. 32).

[137] R. Vallee-Rai and L. J. Hendren. *Jimple: Simplifying Java Bytecode for Analyses and Transformations*. 1998 (cit. on pp. 18, 28).

[138] J. D. Vecchio et al. “String Analysis of Android Applications (N)”. In: *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. 2015, pp. 680–685. DOI: 10.1109/ASE.2015.20 (cit. on pp. 96, 162).

[139] T. Vidas and N. Christin. “Evading Android Runtime Analysis via Sandbox Detection”. In: *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*. ASIA CCS ’14. Kyoto, Japan: ACM, 2014, pp. 447–458. ISBN: 978-1-4503-2800-5. DOI: 10.1145/2590296.2590325. URL: http://doi.acm.org/10.1145/2590296.2590325 (cit. on p. 14).

[140] N. Viennot, E. Garcia, and J. Nieh. “A Measurement Study of Google Play”. In: *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS ’14. Austin, Texas, USA: ACM, 2014, pp. 221–233. ISBN: 978-1-4503-2789-3. DOI: 10.1145/2591971.2592003. URL: http://doi.acm.org/10.1145/2591971.2592003 (cit. on p. 62).

[141] I. Watson. “Watson libraries for analysis”. In: *Main Page* (). URL: http://wala.sourceforge.net/wiki/index.php (cit. on pp. 13, 153).

[142] F. Wei, S. Roy, X. Ou, et al. “Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps”. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’14. ACM. 2014, pp. 1329–1341 (cit. on p. 96).

[143] J. Whaley et al. “Using Datalog with Binary Decision Diagrams for Program Analysis”. In: *Proceedings of the Third Asian Conference on Programming Languages and Systems*. APLAS’05. Tsukuba, Japan: Springer-Verlag, 2005, pp. 97–118. ISBN: 3-540-29735-9, 978-3-540-29735-2. DOI: 10.1007/11575467_8. URL: http://dx.doi.org/10.1007/11575467_8 (cit. on p. 58).

[144] X. Xiao and C. Zhang. “Geometric Encoding: Forging the High Performance Context Sensitive Points-to Analysis for Java”. In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ISSTA ’11. Toronto, Ontario, Canada: ACM, 2011, pp. 188–198. ISBN: 978-1-4503-0562-4. DOI: 10.1145/2001420.2001443. URL: http://doi.acm.org/10.1145/2001420.2001443 (cit. on p. 36).

[145] X. Xiao and C. Zhang. *Scaling Context Sensitive Points-to Analysis by Geometric Encoding*. Tech. rep. oai:CiteSeerX.psu:10.1.1.431.1652. The Hong Kong University of Science and Technology, 2014. URL: http://www.cse.ust.hk/~richardxx/papers/geom-tr.pdf (cit. on p. 36).

[146] X. Xiao et al. “Persistent pointer information”. In: *ACM SIGPLAN Notices* 49.6 (2014), pp. 463–474 (cit. on p. 47).

[147] R. Xu, H. Saïdi, and R. Anderson. “Aurasium: practical policy enforcement for Android applications”. In: *USENIX Security 2012*. Security’12. Bellevue, WA: USENIX Association, 2012, pp. 27–27 (cit. on p. 14).

[148] D. Yan, G. Xu, and A. Rountev. “Demand-driven Context-sensitive Alias Analysis for Java”. In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ISSTA ’11. Toronto, Ontario, Canada: ACM, 2011, pp. 155–165. ISBN: 978-1-4503-0562-4. DOI: 10.1145/2001420.2001440. URL: http://doi.acm.org/10.1145/2001420.2001440 (cit. on p. 58).

[149] L. K. Yan and H. Yin. “DroidScope: seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis”. In: *USENIX Security 2012*. Security’12. Bellevue, WA: USENIX Association, 2012, pp. 29–29 (cit. on p. 14).

[150] C. Yang et al. “DroidMiner: Automated Mining and Characterization of Fine-grained Malicious Behaviors in Android Applications”. In: *Computer Security - ESORICS 2014: 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part I*. Ed. by M. Kutyłowski and J. Vaidya. Cham: Springer International Publishing, 2014, pp. 163–182. ISBN: 978-3-319-11203-9. DOI: 10.1007/978-3-319-11203-9_10. URL: http://dx.doi.org/10.1007/978-3-319-11203-9_10 (cit. on p. 153).

[151] S. Yang et al. “Static control-flow analysis of user-driven callbacks in Android applications”. In: *International Conference on Software Engineering (ICSE)*. 2015 (cit. on p. 88).

[152] S. Yang et al. “Static Control-flow Analysis of User-driven Callbacks in Android Applications”. In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. ICSE ’15. Florence, Italy: IEEE Press, 2015, pp. 89–99. ISBN: 978-1-4799-1934-5. URL: http://dl.acm.org/citation.cfm?id=2818754.2818768 (cit. on p. 96).

[153] Z. Yang and M. Yang. “LeakMiner: Detect Information Leakage on Android with Static Taint Analysis”. In: *Software Engineering (WCSE), 2012 Third World Congress on*. 2012, pp. 101–104 (cit. on p. 13).

[154] Z. Yang et al. “AppIntent: analyzing sensitive data transmission in android for privacy leakage detection”. In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. CCS ’13. Berlin, Germany: ACM, 2013, pp. 1043–1054. ISBN: 978-1-4503-2477-9. DOI: 10.1145/2508859.2516676. URL: http://doi.acm.org/10.1145/2508859.2516676 (cit. on p. 98).

[155] M. Zhang and H. Yin. “AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications”. In: *Proceedings of the 21st Network and Distributed System Security (NDSS) Symposium* (2014) (cit. on pp. 101, 103, 114).

[156] Y. Zhauniarovich et al. “StaDynA: Addressing the Problem of Dynamic Code Updates in the Security Analysis of Android Applications”. In: *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*. CODASPY ’15. San Antonio, Texas, USA: ACM, 2015, pp. 37–48. ISBN: 978-1-4503-3191-3. DOI: 10.1145/2699026.2699105. URL: http://doi.acm.org/10.1145/2699026.2699105 (cit. on p. 153).

[157] Y. Zhou and X. Jiang. “Dissecting Android Malware: Characterization and Evolution”. In: *SP ’12*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 95–109. ISBN: 978-0-7695-4681-0. DOI: 10.1109/SP.2012.16 (cit. on p. 112).

[158] H. Zhu, T. Dillig, and I. Dillig. “Automated Inference of Library Specifications for Source-Sink Property Verification”. In: *Programming Languages and Systems*. Ed. by C.-c. Shan. Vol. 8301. Lecture Notes in Computer Science. Springer International Publishing, 2013, pp. 290–306. ISBN: 978-3-319-03541-3. DOI: 10.1007/978-3-319-03542-0_21. URL: http://dx.doi.org/10.1007/978-3-319-03542-0_21 (cit. on p. 114).

Table of Figures

Table of Tables

Table of Listings

Table of Algorithms

Erklärung zur Dissertation

Hiermit versichere ich, die vorliegende Dissertation ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 18. Januar 2017

_____

(Steven Arzt)