The Soot-based Toolchain For Analyzing Android Apps

Steven Arzt
Fraunhofer SIT
Darmstadt, Germany
Steven.Arzt@sit.fraunhofer.de

Siegfried Rasthofer
Fraunhofer SIT
Darmstadt, Germany
Siegfried.Rasthofer@sit.fraunhofer.de

Eric Bodden
Heinz Nixdorf Insitute at Paderborn
University & Fraunhofer IEM
Paderborn, Germany
Eric.Bodden@iem.fraunhofer.de

Abstract

Due to the quality and security requirements that come with an always-on mobile device processing large amounts of highly sensitive information, Android apps are an important target for automated program analysis. Yet, research on new approaches in this field often requires a significant amount of work to be spent on engineering tasks that are not central to the concrete research question at hand. These programming and debugging tasks can significantly delay the progress of the field. We therefore argue that research in the field greatly benefits from having a universal platform of readily usable components and well-tested fundamental algorithms on top of which researchers can build their own prototypes. Besides decreasing the required engineering effort for each new piece of research, such a platform also provides a base for comparing different approaches within one uniform framework, thereby fostering comparability and reproducibility.

In this paper, we present the Soot framework for program analysis and various highly integrated open-source tools and components built on top of it that were designed with re-usability in mind. These artifacts are already at the core of many research and commercial projects worldwide. Due to the shared platform, results from one tool can not only be used as inputs for the others, but individual data objects can be passed around to form one large API with which one can build new research prototypes with ease.

CCS Concepts •**Software and its engineering →Development frameworks and environments;** *Software libraries and repositories;*

Keywords Soot, Android, Program Analysis, Data Flow, Reproducibility, Infrastructure

ACM Reference format:
Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2017. The Soot-based Toolchain For Analyzing Android Apps. In *Proceedings of 4th IEEE/ACM International Conference on Mobile Software Engineering and Systems, Buenos Aires, Argentinia, May 2017 (MOBILESoft'17),* 10 pages.
DOI: 10.475/123_4

1 Introduction

Smartphone apps are ubiquitous nowadays. According to data from Yahoo, the average user has 95 apps installed on her phone, 35 of which she uses on a daily basis [41]. Many users rely on apps for important daily tasks such as reading their business and private e-mails, mantaining their calendars, or giving them directions while on the road. With these reliance on mobile apps, the quality and reliability of these apps is of great importance. This requires developers to maintain high standards in checking their apps for potential bugs and incompatibilities.

On the other hand, managing all this data on a single device also attracts criminals who try to exploit vulnerabilities in existing apps or deploy outright malicious apps to unsuspecting users. With highly accurate sensors for GPS and acceleration built into the devices, data theft goes beyond classical desktop concerns and allows for full user profiling. Even benign apps can pose significant challenges to user privacy. Developers can gain revenue from apps by embedding advertisement libraries into their code that then take care of showing advertisements to the user. Each display of an ad benefits the developer financially. These libraries, on the other hand, often collect various pieces of information about the user for the purpose of building user profiles for targeted advertisement. This can include location data, the set of installed apps on the phone, and various unique identifiers [19, 36]. The concrete behavior of these libraries is often unknown not only to the end user, but also to the app developer who uses the advertisement library as a black box.

All these challenges require approaches for analyzing software, be it mobile apps as a whole or libraries that are to be embedded into such apps. In this paper, we address static and hybrid approaches as well as dynamic analyses based on instrumentation. Although the concrete analysis goals differ for the various stakeholders (developers, malware analysts, end users), they all have some requirements in common. In most cases, source code is either completely unavailable or missing for at least some portions of the app, such as the advertisement libraries. Consequently, one requires an analysis approach that works on binaries, not just on source code. Secondly, since most analysis tasks are complex, developing a single tool from scratch would incur a prohibitive effort. Therefore, it is paramount that new analyses can be built on top of existing, well-maintained, stable and mature frameworks and toolkits with open interfaces. Many analyses, for example, require binary app code to be parsed, a callgraph to be constructed, or specific data flows to be detected. If basic frameworks for these tasks are available, researchers can focus on their contribution instead of engineering efforts on building blocks outside of their primary research question. This observation leads to another requirement: Tight integration requires common abstractions. For instance, if the tool for reading in Android bytecode were to use a different output format than what is expected as an input by the callgraph-construction system, one would have to spend tedious effort on creating suitable interfaces.

In this paper, we present an ecosystem of analysis tools and frameworks built on the Soot framework for program analysis and transformation [24, 37] as their shared base with the Jimple intermediate representation [38] as the common language level for all analyses. The Soot ecosystem allows researchers and practitioners

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
MOBILESoft'17, Buenos Aires, Argentinia
© 2017 Copyright held by the owner/author(s). 123-4567-24-567/08/06…$15.00
DOI: 10.475/123_4

to quickly develop and evaluate new approaches by building on existing, well-tested building blocks, some of which have been under development for over a decade, long before Android or mobile systems in general even entered the field. The tools presented in this paper have not only been used in a variety of research projects all around the world, but have also become central building blocks of industry projects with major companies. In addition, commercial products such as CodeInspect[1] are based on these tools. Still, the basic libraries presented herein are and will remain open source and free for all to use.

The remainder of this paper is structured as follows. In Section 2, we give an overview over the architecture of the Soot analysis ecosystem. In Section 3, we present Soot itself and the various tools that form the Soot ecosystem. In Section 4, we then show how these tools can be applied to solve various common code analysis problems with regard to Android apps, before we conclude the paper in Section 5.

2 Overview

Figure 1 presents an overview over how the various tools interact to cover different aspects of static analysis for Android. Soot is the central platform on which all tools are based. It provides the front-end for reading in the bytecode and the Jimple intermediate representation on which the analyses are conducted. On top of Soot, FlowDroid is the core technique for computing data flows. All other approaches and tools that require data flows integrate FlowDroid for this purpose. The other tools, however, not only extend FlowDroid, but can also give back data for improving the quality of the data flow analysis. StubDroid, for example, uses FlowDroid to compute data flows inside libraries. FlowDroid can then use these summaries to improve the analysis performance for apps that use the pre-analyzed and pre-summarized libraries.

Figure 2 presents how the tools interact, i.e., which data is produced and consumed by which tool. Since all tools are based on Soot, the figure does not contain explicit edges for this basic technology. FlowDroid is grouped together with StubDroid and ICCTA, because the latter two tools extend FlowDroid. For an external user, they are usually integrated as a package. In combination, the three tools provide a highly efficient static data flow tracker that can deal with inter-component communication and large libraries. Flows from this integrated "meta-tool" are then used in FuzzDroid and Harvester. SuSi, on the other hand, provides the inputs (i.e., the sources and sinks), for the data flow analysis. As we will point out in the subsequent sections, the tools presented in this paper are applicable and widely used beyond data flow analysis, though. Each of the following sections focuses on one of the tools and describes the respective tool as well as its integration into our tool suite.

3 Tools

In this section, we present the toolchain based on Soot that can be used to solve various common analysis problems with regard to Android apps. Section 3.1 focuses on Soot and its Jimple intermediate representation. We further explain the FlowDroid static data flow tracker in Section 3.2. For automatically determining sensitive sources and sinks that can serve as inputs to the data flow analysis, we present the SuSi approach in Section 3.3. When dealing with apps that contain large libraries, to maintain scalability it is advisable to abstract from the concrete library implementations. StubDroid implements an approach that allows for such abstraction without any loss of precision (Section 3.4). While these building blocks focus on intra-component analyses, ICCTA (Section 3.5) extends FlowDroid and Soot to inter-component analysis. For some apps, it is relevant to know under which circumstances a certain statement is executed. If an action, for example, is only performed under dubious conditions, this can indicate targeted malware. FuzzDroid enables such analyses (Section 3.6). In Section 3.7, we present a hybrid approach for extracting runtime values from (potentially obfuscated) Android apps called Harvester. The tool can effectively be used for de-obfuscating apps and rewriting them to improve the results of the other tools presented here. All these approaches are building blocks of one large ecosystem. They are all available as inter-linked open-source projects[2], and have been re-used for custom analyses, both in academic research and for building commercial analysis tools.

3.1 Soot

Soot is the core framework on which all other tools are built. Originally, Soot was designed as a framework for the analysis and transformation of Java programs [37]. At a time when the Java Virtual Machine (JVM) was still purely based on emulation, the main purpose of using Soot was to implement ahead-of-time optimizations via bytecode-to-bytecode transformations. The introduction of JIT compilers into modern JVMs reduced the importance of static ahead-of-time optimizations and Soot was evolved into a more generic analysis tool. In addition to finding performance problems [6, 8, 43], Soot-based analyses are now, among other areas, applied to bug finding [10], as well as the identification of security problems [20, 26] and privacy issues [3, 44]. With the introduction of the Dexpler component [5], Soot gained the ability to process Android's Dalvik file format.

All input code is translated to the Jimple intermediate representation [38], regardless of its original format. This concept allows client analyses to work on different platforms, for example by combining the code of an Android app with the Java-based implementation of the Android framework. This provides a more complete view of the app's execution than what would be possible by representing the app alone. Jimple is a three-address language of low complexity. It features only 15 different types of statements that can contain 29 different types of expressions. In comparison to approaches directly based on a disassembler such as Baksmali [13, 42], this greatly reduces the effort to build an analysis. Additionally, Jimple is typed and provides typed local variables, which frees the analysis designer from having to deal with registers (as they exist in Dalvik code) or even stack operations (as in Java bytecode).

Furthermore, we have started to implement front-ends for other bytecode languages such as the Microsoft Intermediate Language (MSIL) as well [2]. These additional front-ends extend the applicability of Jimple beyond Java and Android with the hope that many existing analyses can be re-used for new platforms with minimal effort. Having a single integrated intermediate representation also allows a client analysis to process code from mixed sources, e.g., an Android app that references a Java library. Cross-platform integration is becoming more and more important with the rise of

[1] www.codeinspect.de

[2] For Harvester, the licensing arrangements have not been finalized yet. Therefore, this tool is not available as an open-source project for now. Please contact us for individual arrangements if you would like to use this tool.

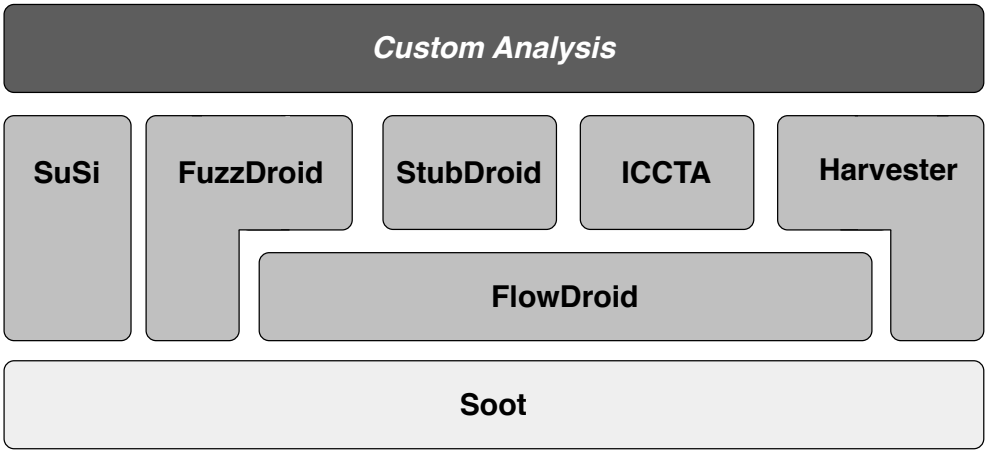

Figure 1. Android Analysis Tools Architecture

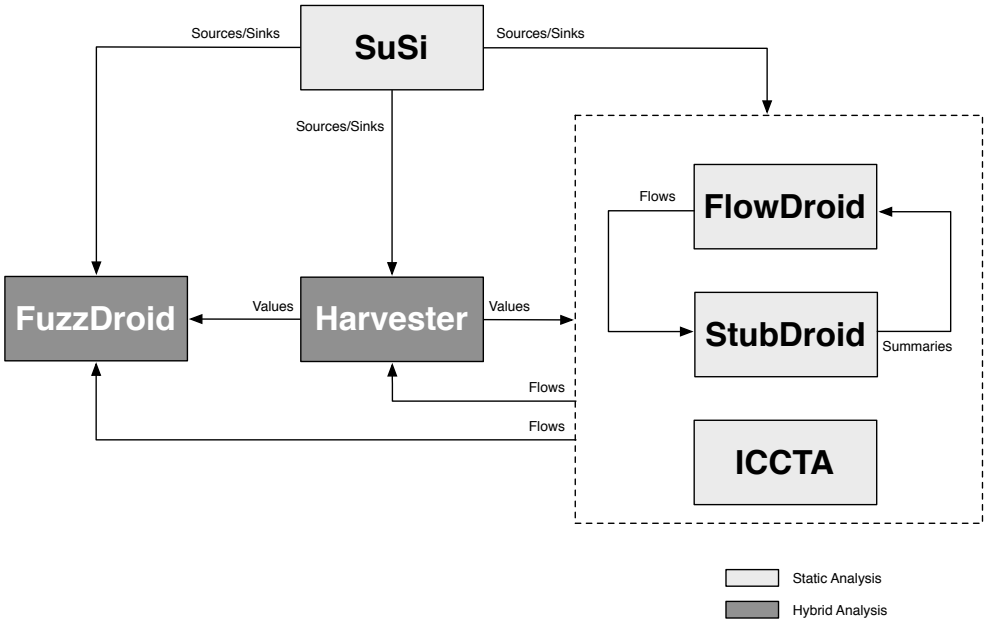

Figure 2. Android Analysis Tools Interaction

the corresponding development suites. We have already observed Android apps and even Android malware written in C# in the wild[3]. Those apps were compiled to MSIL code, which then runs in a different virtual machine alongside Android's normal Dalvik VM (or ART runtime in newer systems).

As an analysis framework, Soot already offers a broad variety of basic analyses that are fundamental for semantically richer client analyses. A client analysis that verifies the legitimacy of outgoing connections may, for example, use Soot's constant value propagation to obtain URLs, IP addresses and host names as string constants inside API method calls without having to backtrack variable assignments on its own. Other analyses may want to use Soot's dead code eliminator to remove code that is left in the app for debugging purposes, but that is never executed in production, because it is control-flow dependent on a Boolean variable that is always set to `false`. The optimizations that stem from Soot's past as a bytecode optimization framework can be very useful for reducing the complexity of the target code before conducting the actual static analysis. Furthermore, Soot provides efficient solvers for popular analysis frameworks such as the monotone framework or (through Heros [7]) the IFDS framework by Reps, Horwitz, and Sagiv and the IDE framework by the same authors.

[3] Package name `com.tinker.gameone`, MD5 `3ae3cb09c8f54210cb4faf7aa76741ee`

3.2 FlowDroid

FlowDroid [3] extends Soot with a highly precise static data-flow tracker for Android and Java. The main use case for which FlowDroid was developed was to find privacy issues in Android apps.

Data flows are, however, also required for many other problems. Integrity violations and SQL Injections, for example, are dual problems and can thus directly be solved using FlowDroid as well.

FlowDroid's data flow tracking problem is formulated in the IFDS framework. As its main abstraction, FlowDroid uses access paths. This allows the analysis to not only track taint state on variables and fields along the app's control flow graph, but more precise sequences of field dereferences. If the code, for example, writes sensitive data into `a.b.c.d`, but later leaks a value derived from `a.b.c.e`, this will not lead to a false positive in FlowDroid. Several optimization techniques including a custom-built IFDS solver, allow the data flow tracker to retain high performance and scalability despite its precision. Furthermore, FlowDroid does not rely on Soot's standard alias analysis, because this analysis follows the traditional binary pattern of checking whether two objects potentially alias. For the data flow analysis, on the other hand, whenever a heap object is tainted, all aliases must be enumerated and propagated alongside the original taint (for a detailed discussion, see [1]). Such a requirement does not fit the traditional binary model. Furthermore, traditional alias analyses are limited to one base object and one field which is below the precision of arbitrary-length access paths. Therefore, FlowDroid provides its own integrated alias analysis, which is also formulated as an IFDS problem based on access paths just like the core data flow tracker. This approach allows FlowDroid to not only retain the same precision for alias and data flow analysis, but also schedule both problems on the same multi-threaded infrastructure.

Besides being used as a library for computing data flows, FlowDroid can also serve as a testbed for new algorithms and ideas. In fact, data flow analysis builds on a broad variety of techniques such as alias analysis, points-to analysis, and callgraph analysis. More often than not, those basic building blocks are researched and evaluated in isolation. As a consequence, the implicit assumptions of these approaches, e.g., regarding the number of queries posted to an alias analysis, or the distribution of variables for which such queries are posted, remain unclear and unvalidated. If another researcher needs to integrate such a technique, she often needs to resort to trial-and-error when evaluating which approach is best suited for her needs.

Thanks to FlowDroid's open architecture, such building blocks can be tested and evaluated in a real-world scenario on massive data. If a new alias analysis, for example, is integrated as an implementation of FlowDroid's `IAliasingStrategy` interface, it can be used for running a data flow analysis on a large number of real-world Android apps of substantial size and complexity. Delivering good results in such a scenario greatly improves the confidence in the presented approach. Since FlowDroid is based on the Soot framework, integrating building blocks that are also based on Soot is easily possible. Furthermore, having a common framework in which to try out new approaches allows for easy comparison between techniques and in relation to FlowDroid's already existing default implementations. One alias analysis that has already used FlowDroid as a testbed is Bommerang [35], which lead to new insights on the analysis' tradeoff between precision and performance.

A second interface abstracts from FlowDroid's handling of native method calls. By default, the data flow tracker relies on a small set of hand-written rules for the most common native method calls that occur inside the Java and Android base libraries. If other researchers would like to contribute a data flow analysis for native code that is capable of modelling calls to the Java Native Interface (JNI), FlowDroid is the ideal testbed for such an analysis. With only a small set of methods to be implemented, the contributing researcher can quickly try out her analysis on a large number of real-world Android apps from the Google Play Store that make use of native code. Various automated approaches for identifying and downloading such candidate apps have already been proposed as well [32, 39].

In fact, the core FlowDroid data flow tracker is platform-agnostic and abstracts away from all platform-specific models and techniques using open interfaces. It provides default implementations of these interfaces that model the Java semantics. Outside of FlowDroid's core, a set of extensions provides the core with the semantics of the Android plattform and additional algorithms such as the callback collection explained above. This shows that FlowDroid is flexible enough to accomodate a variety of different analysis targets and algorithms. Even for the core IFDS solver, multiple implementations exist (FlowDroid's own *Fast Solver*, one based on Heros [7], and a flow-insensitive variant of the Fast Solver). More can be added to explore new trade-offs between precision and scalability.

3.3 SuSi

Before one can conduct a data flow analysis, one must specify the set of sources and sinks. Sources are all methods which the app can use to obtain sensitive information; only those values are to be tracked by the data flow analysis. Sinks are the only methods for which an alarm should be raised if they are supplied with sensitive data. Therefore, missing a source or a sink can result in critical data flows being missed, regardless of the quality of the concrete data flow solver being used. When the analyst aims at finding security vulnerabilities or privacy violations, such an incomplete set of sources and sinks can lead to a false sense of security.

Given the more than 140,000 methods already in Android 4.2, manually assembling a complete list of sources and sinks is practically infeasible. In fact, as we have shown in our previous work on SuSi [31], popular analysis tools that rely on such hand-crafted lists miss important sources and sinks that allow an attacker to leak data from a device without being detected. SuSi, on the other hand, uses machine learning to automate the process of identifying sources and sinks from a binary distribution of the Android framework. The approach uses a small subset of hand-annotated sources and sinks to train a classifier that then decides on the remaining methods in the framework. In our experiments, we showed this approach to yield a precision and recall of more than 92%. In fact, there are many more sources and sinks in Android than usually considered. We have identified hundreds of sources and sinks in Android 4.2. Notably, identifying sources and sinks in the Android framework alone is not sufficient. Many libraries such as the Google Chromecast SDK come with their own methods for obtaining sensitive data or for sending data to untrusted receivers. One key advantage of an automated approach such as SuSi is that it requires a mere re-run of the approach on a different JAR file. In many cases, it is not even required to extend the hand-annotated training set, although this is possible in case the results on the new library are not satisfactory.

Having a comprehensive set of sources and sinks, however, also comes with some disadvantages. With such a large number of sources and sinks, analysts can easily be overwhelmed. Additionally, when the data flow analysis is run with all sources and sinks in the whole Android framework, this can also pose challenges for

the scalability of the data flow tracker. Every return value of each source method needs to be tracked through the complete app, at every line at which it is in scope. Secondly, with so many sources and sinks, it can be time-consuming to identify the really important data flows from a large number of flows that are technically correct but not relevant for the analysis task at hand. To alleviate this problem, SuSi provides a second step that automatically categorizes the sources and sinks after they are discovered. Our 14 categories for sources include, among others, account data, contact data, data obtained from the network, location data, and unique identifiers. The 17 sink categories include, among others, data written to the calendar or contact database, to the network, or being sent out via SMS or MMS. The analyst can use these categories to only pass those sources that are of interest for her concrete goals to the data flow analysis. Excluding all other categories greatly redcues the computational effort and the required amount of memory.

SuSi integrates with FlowDroid via configuration files. The source and sink list generated by SuSi can directly serve as the input for the FlowDroid data flow tracker. Since SuSi and FlowDroid share data structures and parsers, they can both also reade a variety of other formats, including the API lists generated by the PScout project [4] and the RIFL specification format [15] used by the Cassandra certifying app store [27].

3.4 StubDroid

Java programs and Android apps are often implemented on top of large libraries. These libraries spare the developer from implementing common tasks such as cryptography, file handling, error reporting, and graphics rendering on his own. While relying on libraries is convenient and good software engineering practice, it also poses challenges for static analysis. Especially in the context of Android apps, the library is compiled into the same APK file as the user code, effectively leaving the static analysis with one large package. This package consists mainly of library code, and hence, most of the analysis effort is spent on the libraries, not on the actual user code. Furthermore, since many apps share the same libraries, and those libraries are packaged into all of these apps, the same library code is analyzed over and over again. This effort significantly increases the time and memory required for the analysis.

To alleviate this problem, analyses can rely on explicit library models instead of analyzing the library code. While this approach reduces the time and memory consumption, it poses new challenges in obtaining the required library models. Placing this burden upon the user is a common approach [16–18, 22, 28, 45], but not acceptable in practice. Even if the average user is assumed to have the technical understanding required for constructing the models, defining them manually is a major effort. Relying on user-defined models also leads to a questionable completeness of the whole approach if the user fails to provide (complete) models for some library methods. Therefore, some data flow analyses [21] instead over-approximate the library models through rules of thumb such as "The return value of a method is always tainted if at least one of its parameters is tainted". These models are very coarse-grained and do not take the specific behavior of the particular method at hand into account. Consesquently, false positives can arise. Even worse, the original performance and memory problem can appear again due to over-tainting.

To alleviate this problem, StubDroid automatically analyzes library classes and creates precise *per-method* data flow summaries, which can then be plugged into a FlowDroid data flow analysis. Instead of having to analyze the concrete library implementation, FlowDroid can then abstract from the library code at the level of the public interface. StubDroid rules take the form "when method m() is called with the first parameter tainted before the call, the return value is tainted after the call". Such a rule requires taints to only be propagated across a single data flow edge (i.e., the summary edge), regardless of the original size of the implementation of method `m()`. This approach is especially useful for deep call hierarchies which are common in the Java and Android base libraries. In the Java collection APIs, using StubDroid summaries can reduce the computation time by about 80% and even allow for those cases to be analyzed for which the analysis would otherwise time out or exhaust its memory allowance. Note that StubDroid analyzes each library method in isolation, which can further prevent the common problem of exponentially-growing taint sets. In other words, all taints that are generated during the analysis of one method are confined to that method. Only the summarized models are later combined.

Note that StubDroid summaries are tailored to individual methods and thus avoid the over-approximation problem of the rules of thumb. They are also based on access path just like the FlowDroid data flow analysis. Consequently, there is no loss of precision when substituting a library implementation by a StubDroid model during a FlowDroid analysis. Furthermore, StubDroid also summaries aliasing information for libraries with the same level of precision. In fact, since StubDroid not only extends FlowDroid with library summaries, but also internally uses FlowDroid to generate the summaries, it provides serialization and summarization for FlowDroid's aliasing infrastructure. In summary, StubDroid is not only an important extension to improve the scalability of FlowDroid, but can also serve as a building block for other Soot-bases analyses that need to deal with large libraries, either with regard to data flow or with regard to aliasing.

3.5 ICCTA

The FlowDroid data flow tracker focuses on finding data flows inside a single Android component. Consequently, if data is received from the outside, e.g., through an incoming intent, or leaves the component, e.g., through an outgoing intent, these points must be modeled as sources or sinks, respectively. For obtaining a complete picture of an app, such flows are, however, not satisfactory. When looking for privacy issues or security flaws, flows that only occur inside an app, regardless of whether they cross component boundaries or not, are oftentimes not the primary concern. Consequently, FlowDroid must be extended to combine the various detected intra-component flows according to the control flow edges between the components. This leads to two research challenges: Firstly, the inter-component control flow edges must be identified, and, second, the flows must be combined.

For enumerating the inter-component flows, various tools have been built on top of the Soot framework, including EPICC [30], IC3 [29], and Harvester which we present in Section 3.7. For integrating the flows based on these control flow edges, there are two general approaches. The first approach, which is taken by ICCTA [26], is to rewrite the app so that all inter-component communication is replaced with simple Java method calls. The resulting app then only consists of one large component that can be analyzed by FlowDroid. In other words, ICCTA can be thought of

as a pre-analysis step for FlowDroid. The second, alternative approach, implemented in DidFail [23], instead first analyzes each component in isolation and then, outside of the data flow tracker, combines the discovered flows based on the inter-component control flow edges. Both approaches are well integrated into the Soot and FlowDroid tool chain and both are well-suited for the purpose of inter-component data flow analysis. The ICCTA approach, however, has the advantage of generating a single integrated callgaph, which is then also available to client analyses that build upon FlowDroid. With this approach, the client analysis can abstract from Android's component-based architecture and concentrate on its actual analysis problem. With the DidFail approach, on the other hand, the integration must happen on a higher level, i.e., on the semantic level of per-component analysis results.

Note that inter-component communication is technically equivalent to inter-app communication in Android. The only difference lies in how the operating system performs access control checks. Therefore, both the ICCTA and the DidFail approach can also be applied to sets of communicating apps.

3.6 FuzzDroid

When applying dynamic analysis to Android apps, code coverage is a known problem [12] with many approaches not being able to exercise more than 30% of the app's code. One contributing factor, especially in the context of malware apps, is that the behavior of the app under analysis can depend on its execution environment. A malware app may, for instance, only send out unsolicited text messages to expensive premium-rate telephone numbers when it is not running inside an emulator. Other apps check whether they are running on phones with sim cards registered in specific countries, or only perform certain tasks at night-time. For successfully employing dynamic or hybrid analyses, the target device must be configured accordingly. Due to the high number of different types of environment variables in Android, it is infeasible to try out all possible combinations of these variables and look for changes in app behavior. In Android, an app can query for SIM card properties (operator name, country code, etc.), battery charging level, device properties (screen resolution, device manufacturer and model, etc.), available types of Internet connectivity, and many more parameters.

As a solution to this problem, we propose FuzzDroid [34]. The goal of FuzzDroid is to report concrete environment values under which a user-defined code position inside the app is executed. FuzzDroid combines static code analysis with dynamic fuzzing techniques to generate candiate environments, check them at runtime, and then use the learned runtime behavior to further improve the next candidate environment. This process continues until either the requested code location has been reached, or no further progress can be made. The latter can happen if the target location, for example, is indeed unreachable regardless of the environment.

FuzzDroid can therefore serve as an important building block when implementing instrumentation-based dynamic analyses using Soot. When the API-to-value mappings from FuzzDroid are known, the analyst can either reconfigure her device to match the particular configuration, or can use Soot's code manipulation techniques to rewrite the app such that a suitable fake environment is injected at runtime. With the latter technique, the analyst replaces all environment API calls in the app with constant values that are known to be suitable for reaching the code position of interest. This bytcode modification approach is suitable even if the property as such cannot be changed without low-level modifications to the emulator infrastructure.

Furthermore, FuzzDroid provides an infrastructure for building reliable hybrid analyses with Soot. The core idea is that a (possibly resource-intensive) static analysis runs on a desktop computer or server, while the app is executed on an emulator or a real phone. The static analysis results can be used to more effectively configure and steer the app execution, while the runtime behavior can be tracked and evaluated to more precisely steer the static analysis. With every run, more data is available, until the process ideally converges to the solution of the overall analysis task. In FuzzDroid, the static analyses that generate new candidate environments are called *value providers*. The dynamic runtime environment that is instrumented into the app provides the value providers with runtime information such as a dynamic callgraph, the runtime values of certain requested variables or fields, and exception data in case the app has crashed due to an unhandled exception. Additionally, FuzzDroid monitors the app for certain events such as dynamic code loading. In that case, it extracts that `dex` file that is being loaded and sends it back to the analysis running on the desktop computer. The static analysis can then merge this `dex` file into its Soot instance and improve its recall.

We believe that FuzzDroid, while not yet a library that can be used out of the box, provides valuable patterns and examples of how to build efficient hybrid analyses using Soot. This spares the analysis developer from having to deal with technical details such as communication between the app and the host computer, or implementing the correct transformations that gather all the information required to build a dynamic call graph.

3.7 Harvester

While analyzing Android apps already poses challenges in itself, many apps are further obfuscated to deliberately hinder both manual and automatic analysis. In the context of malicious apps, the reason for code obfuscation is obvious, as app store providers are continuously monitoring their stores in an attempt to identify and remove such malicious code. Consequently, the malicious app can only generate substantial revenue for the miscreant until its behavior is detected by the store provider. After detection, existing installations of the malware on victims' devices might still be active, but it becomes considerably harder to infect new phones. In fact, Google even provides a feature that allows store officials to remotely remove known malicious apps from phones. However, code obfuscation is not limited to malicious apps. Benign, but sensitive software such as banking apps use very similar obfuscation techniques to make it harder for an attacker to discover potential security flaws inside the app. Yet other app developers employ code obfuscation for protecting their developers' intellectual property such as algorithms from being reverse engineered by competitors.

In total, obfuscation techniques are increasingly used. In the most basic case, only the class, method, and field names are changed. Proguard is a tool for such basic obfuscation which is freely available and even shipped together with the official Android SDK. App developers are encouraged to apply it to their app before uploading it to the Play Store. This name obfuscation does not pose a challenge to most static analysis approaches, because they do not expect these names to carry semantics. More involved obfuscation techniques, however, can render a purely static analysis practically impossible.

Some commercial protection tools such as APK Protect and DexGuard automatically rewrite an app to not call methods directly as normal invocations, but instead use reflection. The class and method names that are passed to the Java reflection API as strings are not plain text constants, though. Instead, they are computed at runtime. Most commonly, the names are encrypted in some way (techniques range from simple bit-shifting techniques to proper encryption using algorithms such as AES), and are only decrypted at runtime using a key derived from a computation. In other words, the mapping from call site to callee is only available at runtime, sometimes even only on demand, i.e., with the class and method names only being decrypted right before they are needed for a call. Without this information, a static analysis can only generate a massively over-approximate callgraph that essentially maps every call site to every callee with a matching list of parameter types. Clearly, such an approach is highly imprecise and impractical for many client analyses.

To alleviate the problem, we have implemented a hybrid (static and dynamic) approach for reconstructing runtime values from obfuscated apps. Our tool Harvester first uses static slicing to identify all binary instructions that contribute to the computation of a given runtime value. Such a value can, for example, be the class name passed to `Class.forName()` in a reflective method call. In this case, the respective parameter of `Class.forName()` is called a *logging point*. The concrete value that is computed at runtime is called a *value of interest*. Harvester extracts all identified contributing statements (i.e., the slice) and places them into a new app where they can be executed in isolation. This app is then installed and run on an emulator or a real phone. It directly calls the slice code and reports the computed value back to the computer on which Harvester runs. To enable other analyses such as FlowDroid to process obfuscated apps, Harvester can rewrite the app to directly integrate the discovered runtime values. The logging points are replaced by string constants containing the computed values of interest. In the case of Android inter-component communication, this rewriting allows tools such as EPICC or IC3 to obtain more precise inter-component callgraphs, which in turn allow for much more precise data flow tracking with FlowDroid and ICCTA. For reflective method calls, Harvester replaces the calls to the Java reflection API with direct method calls to the actual target methods. This rewriting effectively de-obfuscated the apps. Afterwards, FlowDroid can construct a callgraph for the app as usual.

As already discussed in Section 3.6, some apps behave differently under different execution environments. Popular checks inside environment-dependent apps include the country of the user's SIM card or her device ID. When computing runtime values, the analyst is often interested in enumerating all values that can be computed at a given code position, not only those, that happen to occur in her concrete setup. Therefore, when Harvester detects a conditional based on an environment value, it explores both branches. For checking whether a conditional depends on an environment value, it uses static data flows from FlowDroid. It then rewrites the conditional to reference a global executor variable. When the app runs on the emulator, Harvester first sets the executor variable to `true` to explore the *then* branch of the conditional. When the values have been computed, Harvester restarts the app with the executor variable set to `false` to explore the *else* branch of the conditional. This makes sure that all possible outcomes of the conditionals are considered. Note that the use of static data flow information is not an issue even in the presence of features that the static analysis cannot reason about, because Harvester can be run incrementally: It first analyzes the app and discovers some values. Afterwards, it rewrites the app to remove reflective method calls and and integrate constants. This makes it then easier for FlowDroid to discover more flows, and Harvester can use this improved data flow information on its next run. Each run leads to more runtime values being discovered. This approach shows how FlowDroid and Harvester mutually contribute data to improve the overall result. Since both tools are based on Soot, such integrations are easily possible without loss in precision or complex data type conversions.

4 Android Code Analysis Problems

In this section, we describe how the tools presented in Section 3 can be used to solve a variety of common Android code analysis problems when dealing with Android apps. Table 1 lists the most common problems together with the tools that solve them. Note that for the more involved problems, multiple tools work together to provide the respective feature. This again shows the impact of a highly integrated ecosystem in which tools can be built on top of each other to solve even complex problems with reasonable effort.

Code Analysis Problem	Tools
Bytecode Instrumentation	Soot
Callgraph Construction	Soot + FlowDroid
Points To Set Construction	Soot + FlowDroid
Data Flow Analysis	Soot + FlowDroid
Runtime Value Extraction	Soot + FlowDroid + Harvester
Runtime Environments	Soot + FlowDroid + FuzzDroid
Library Summaries	Soot + FlowDroid + StubDroid
Source / Sink Detection	Soot + SuSi

Table 1. Android Code Analysis Problems and Tools To Solve Them

4.1 Bytecode Instrumentation

The heritage of originally being a compiler framework allows Soot to integrate both capabilities for analysis and transformation in a single tool, a feature that is lacking from other well-known analysis platforms such as Wala [40] and OPAL [14]. Therefore, not only static analyses can be constructed with Soot, but also instrumentation-based dynamic analyses [44] and runtime policy enforcement [33] are possible within the very same framework. This dualism of analysis and transformation is also a very important aspect for researchers working on hybrid analyses that combine static and dynamic aspects. Especially for security checks, purely static analysis is often not able to certify the absence of policy violations without accepting a large number of false positives. Purely dynamic analysis, on the other hand, incurs a prohibitive runtime overhead. A hybrid approach can alleviate the problems of both techniques by combining them. In that case, a static pre-analysis, for example, can identify those parts of the app that can never violate the policy. Those parts then be excluded from the dynamic performance to improve runtime performance [6, 8, 9]. With Soot, the pre-analysis and the instrumentation for the runtime checking code can operate on the same representation of the program, without requiring the analysis designer to convert representations or

even handle different levels of abstraction between tools. Beyond performance improvement, hybrid analyses on an integrated platform can also allow for new combinations of static and dynamic techniques. In DroidForce [33], for example, we use static analysis with FlowDroid to compute the intra-component data flows and integrate them into the app. At runtime, we then combine the intra-component data flows with the exact inter-component data flow edges that are observed to build a full data flow path through the app. With these full flows, DroidForce can then enforce complex policies that also include time and state, e.g., "No more than three SMS messages containing contact data may be sent per hour to the same telephone number after 8pm". Though this particular example is highly artificial, it shows the combination of static analysis results with dynamic state. Concerning the data flow analysis as such, the approach of partial static pre-computation that is then extended with precise dynamic inter-component callgraph information is an alternative to the pure static inter-component analysis approach explained in Section 3.5 of this paper, which enumerates flows in general.

4.2 Callgraph & Points To Construction

FlowDroid precisely models the Android lifecycle and is thus commonly used for building callgraphs of Android apps. Technically, FlowDroid does not provide a callgraph-algorithm on its own. Instead, it bridges the gap between Soot's existing callgraph construction framework SPARK [25], which was originally designed for Java programs, and the execution model of Android, which is much more expressive than Java's. In Java, programs contain a single `main()` method to which, at startup time, the JVM passes control using a reflective call [11]. Afterwards, the program has little interaction with the virtual machine's logic. Android apps, however, are plugins into the Android framework, and follow its control-flow, defined by numerous lifecycles. Developers implement classes that are inherited from framework-supplied superclasses. This thight coupling allows Android to not only suspend and resume an app whenever necessary, but also to notify apps of system-wide events such as battery shortage or an incoming text message. A static analysis for Android can only be precise, and gain sufficient completeness, if it respects the exact set of possible control flows through the app, even though these control flows are orchtestrated by code that resides outside the app. FlowDroid bridges the concepts of Java and Android by generating an app-dependent dummy `main()` method that emulates the interactions between the app and the operating system. While this `main()` method is not a full, executable implementation, it is equivalent to the real system behavior with respect to callgraph construction, points-to analysis and control-flow computation. Consequently, to obtain a precise and largely complete callgraph of the app, the analysis designer can feed FlowDroid's dummy `main()` method into any of Soot's callgraph algorithms. This feature is useful for many derived works, even if they do not use FlowDroid's data flow analysis as such.

For constructing the dummy `main()` method, FlowDroid needs to analyze various types of files, including source code, the Android manifest file, layout XML files, and the app's resource database `resources.arsc`. Thanks to its modular architecture, FlowDroid offers this broad variety of analyzers and parsers to other analyses that build on FlowDroid as well. The analysis designer can, for example, discover a reference to a user-interface control in the code. Such references are merely numeric IDs in compiled Android apps. FlowDroid provides all necessary features for identifying the respective UI control. In the data flow tracker, we use this capability to distinguish between normal input fields and password fields, which are usually more sensitive. Other analyses can build more comprehensive UI models, for instance based on semantic information such as field labels, without having to re-implement the layout file parsers.

When creating the dummy `main()` method, a major challenge lies in accurately identifying callbacks and associating them with the correct host component that receives them at runtime. If this mapping is incomplete, the `main()` method misses calls to callback methods. Consequently, the callgraph constructed with this method as its entry point is incomplete as well and potentially interesting code parts are marked as unreachable. If the mapping, on the other hand, is imprecise, e.g., due to over-approximation, the `main()` method contains spurious calls, which, in turn, lead to over-sized callgraphs. If an analysis uses such an over-sized callgraph, it must follow a great number of irrelevant edges and will usually not scale. Therefore, we took great care to accurately model callbacks in FlowDroid. Note that Android app developers cannot only register callbacks in the code, but also declare them directly inside the respective layout XML file. This approach is often taken for button click handlers, for example. Therefore, the callback analysis must relate the callbacks declared in the app's layout XML files to the objects available in the app's code. This is a multi-staged process of associating IDs, resource database entries, and XML data. If a client analysis uses FlowDroid to construct a callgraph, all of this complexity is abstracted away inside the FlowDroid library.

Overall, Soot and FlowDroid provide important building blocks for analyzing Android apps. They allow the analysis desinger to focus on its main objective, while simply re-using existing components for building blocks such as callgraph construction, manifest file parsing, UI analysis, or callback analysis. FlowDroid's API is a direct extension to Soot's, and uses Jimple objects. Therefore, a callback method returned by FlowDroid's callback analysis, for example, is an instance of `SootMethod`. The client analysis can directly use this method object just like any other method in Soot, without having to convert any representation between tools.

4.3 Data Flow Analysis

FlowDroid can be used as a standalone tool for data flow analysis as well as through its public API. In both cases, the analyst (or client analysis) must provide the APK file to be analyzed and a set of sources and sinks. All data obtained from the sources is tracked through the program. If data derived from any such tainted object reaches a sink, it is logged as possible data leak. By default, the discovered leaks are printed to the console and can optionally be written into an `xml` file. With the API, the client analysis gets direct access to the Soot data objects for the source / sink method and class, as well as, if the respective feature is enabled, all Soot statements on the path from the source to the sink. Especially the latter can be very useful if further analysis is to be performed on the computed data flow paths. Due to the tight integration of Soot and FlowDroid, no external conversions or mappings are required.

The most simple approach for specifying sources and sinks is to use the text-based format that FlowDroid shares with SuSi. The analyst can even copy the interesting subset of the SuSi output into the FlowDroid source and sink file and then run the data

flow solver without any further configuration. For more advanced cases, FlowDroid offers an XML format in which the analyst can precisely pinpoint which access paths shall be tainted at which source and which access paths arriving at which sink shall be considered leaks. Programatically, the client analysis can also easily extend FlowDroid with completely different notions of sources and sinks. By default, sources and sinks are assumed to be access paths based on parameters or return values for method calls. For the core data flow solver, there is not such requirement. Therefore, a client analysis can pass arbitrary pairs of statements and Soot locals (i.e., variables) to the solver as sources. For sinks, a custom provider implementation receives all tainted values that appear at any statement and can then decide whether to consider this a leak or not. Harvester, for example, uses this flexibility to consider conditionals as sinks when looking for environment-dependent conditionals inside the app code.

4.4 Runtime Value Extraction

For obtaining runtime values from an Android app, the analyst needs to define a set of logging points (see Section 3.7). Together with the APK file, this set serves as the input for the Harvester tool. Internally, Harvester utilizes Soot and FlowDroid to read in the app, enumerate the conditionals that depend on environment variables, compute the slices, build the executor apps, and run them on the emulator. As an output, Harvester provides a database with values for each logging point that the analysis has reached at runtime. Client analyses can then process this database without any dependencies on Soot or Harvester. More conveniently, Harvester can, however, also be used as a library. In that case, the client analysis can directly interact with Soot objects for further processing.

4.5 Runtime Environments

For obtaining a runtime environment under which a given Android app executes a given logging point, the analyst only needs to provide the app as well as the set of logging points that shall be triggered. FuzzDroid automatically takes care of all the required tasks such as instrumenting the app, installing it on the emulator, executing it, and correctly conducting the feedback loop of generating new candidate environments and trying them out on the emulator. The results are written into a file. Since the approach is extensible, users can easily write new value providers that supply FuzzDroid with additional values to become part of new candidate environments in case the existing providers are not sufficient for the case at hand. Since FuzzDroid is built on Soot and FlowDroid, all value providers can directly work on Soot's data objects, which are also directly linked to the data flows provided by FlowDroid. This allows value providers to also take flow information into account where required.

4.6 Library Summaries

Library summary handling consists of two parts: Firstly, the summaries must be computed. Afterwards, they can be applied to data flow analysis or FlowDroid's integrated context-sensitive alias analysis. For computing summaries, StubDroid only needs a binary implementation of the target library. The generated summaries take the form of `xml` files, one per class. When running FlowDroid with these summaries, StubDroid's implementation of the `taint wrapper` interface takes care of loading the correct `xml` files on demand and mapping their contents to the concrete incoming taint abstraction. Users of FlowDroid merely need to enable the StubDroid integration and specify the path to the StubDroid library and the `xml` files.

4.7 Source / Sink Detection

For detecting sensitive sources in a given version of the Android framework or any other Java-based library, the analyst first needs to obtain an implementation of that library. In the case of Android, the `android.jar` files shipped with the official Android SDK are not sufficient, because they only contain method stubs that always throw an exception when called. These stubs are designed such that new apps can be linked against them during development. When the app is later executed on a real device or an emulator, it can then use the actual implementation. On Android, the framework classes are already mapped into the execution host that the operating system provides for the app. For identifying sources and sinks, this means that the analyst must first extract this real implementation from an emulator or a device. Once such a complete JAR file is available, it can be passed to SuSi.

5 Conclusions

In this paper, we have presented an overview over static and hybrid analysis tools for Android apps based on the Soot compiler framework. We have discussed how these tools interact on one common platform. For the external researcher that aims to design and implement her own analyses on top of this platform, the existing tools provide ready-to-use implementations of common building blocks such as callgraph construction, data flow analysis, library summary generation, and runtime value analysis. Having such a platform at her disposal, the researcher can focus on her actual contribution without having to spend effort on infrastructure technique afar from her field of research. Using a common infrastructure also greatly improves the reproducability and comparability of the research projects built on top of them. It offers a broad field of experimentation for new algorithms and techniques by providing open interfaces for integrating custom solutions. A new alias algorithm, for example, can then be used as part of, e.g., a full data flow analysis on large real-world apps without the researcher having to care for all other parts of the analysis. Consequently, the work presented in this paper aims at improving the speed and efficiency of scientific work in the area of Android program analysis.

Acknowledgments

This work was supported by the German Federal Ministry of Education and Research (BMBF), by the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP, by the German Research Foundation through the Projects RUNSECURE and CROSSING, through a Fraunhofer ATTRACT grant and through the Heinz Nixdorf Foundation.

References

[1] Steven Arzt. 2017. *Static Data Flow Analysis for Android Applications.* Ph.D. Dissertation. Technische Universität Darmstadt.

[2] Steven Arzt, Tobias Kussmaul, and Eric Bodden. 2016. 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, 1–6.

[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 259–269.

[4] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 217–228.

[5] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. 2012. 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)*. ACM, New York, NY, USA, 27–38. DOI: http://dx.doi.org/10.1145/2259051.2259056

[6] Eric Bodden. 2010. Efficient Hybrid Typestate Analysis by Determining Continuation-equivalent States. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, USA, 5–14. DOI: http://dx.doi.org/10.1145/1806799.1806805

[7] Eric Bodden. 2012. 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)*. 3–8.

[8] Eric Bodden, Laurie Hendren, and Ondrej Lhoták. 2007. A staged static program analysis to improve the performance of runtime monitoring. In *Proceedings of the 21st European conference on Object-Oriented Programming*. Springer-Verlag, 525–549.

[9] Eric Bodden and Patrick Lam. 2010. *Clara: Partially Evaluating Runtime Monitors at Compile Time*. Springer Berlin Heidelberg, Berlin, Heidelberg, 74–88. DOI: http://dx.doi.org/10.1007/978-3-642-16612-9_8

[10] Eric Bodden, Patrick Lam, and Laurie Hendren. 2008. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 36–47.

[11] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders. In *ICSE '11: International Conference on Software Engineering*. ACM, 241–250. http://www.bodden.de/pubs/bss+11taming.pdf

[12] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet?(E). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 429–440.

[13] Anthony Desnos. 2011. Androguard. *URL: https://github.com/androguard/androguard* (2011).

[14] Michael Eichberg and Ben Hermann. 2014. A Software Product Line for Static Analyses: The OPAL Framework. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis (SOAP '14)*. ACM, New York, NY, USA, 1–6. DOI: http://dx.doi.org/10.1145/2614628.2614630

[15] Sarah Ereth and Heiko Mantel. 2015. *Towards a Common Specification Language for Information-Flow Security in RS 3 and Beyond: RIFL 1.0–The Language*. Technical Report TUD-CS-2014-0115. MAIS, Computer Science, TU Darmstadt. http://www.mais.informatik.tu-darmstadt.de/WebBibPHP/papers/2014/RIFL1.0-TechnicalReport-Revision1.pdf

[16] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 2013. *Apposcopy: Semantics-Based Detection of Android Malware*. Technical Report. Stanford University. submitted for publication.

[17] Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster. 2009. Scandroid: Automated security certification of android applications. *Manuscript, Univ. of Maryland, http://www. cs. umd. edu/avik/projects/scandroidascaa* 2, 3 (2009).

[18] Michael I Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. 2015. Informationflow analysis of Android applications in DroidSafe. In *Proc. of the Network and Distributed System Security Symposium (NDSS). The Internet Society*.

[19] Michael C Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. 2012. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*. ACM, 101–112.

[20] Philipp Holzinger, Ben Hermann, Johannes Lerch, Eric Bodden, and Mira Mezini. 2017. Hardening Java's Access Control by Abolishing Implicit Privilege Elevation. In *2017 IEEE Symposium on Security and Privacy (Oakland S&P)*. IEEE, IEEE Press. To appear.

[21] Wei Huang, Yao Dong Ana Milanova, and Julian Dolby. 2015. *Scalable and Precise Taint Analysis for Android*. Technical Report. Technical report, Department of Computer Science, Rensselaer Polytechnic Institute.

[22] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, and Junbum Shin. 2012. ScanDal: Static Analyzer for Detecting Privacy Leaks in Android Applications. In *MoST 2012: Mobile Security Technologies 2012*, Hao Chen, Larry Koved, and Dan S. Wallach (Eds.). IEEE, Los Alamitos, CA, USA. http://ropas.snu.ac.kr/scandal/

[23] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. 2014. 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, 1–6.

[24] Patrick Lam, Eric Bodden, Ondrej Lhotak, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*.

[25] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java Points-to Analysis Using Spark. In *Compiler Construction*, Grel Hedin (Ed.). Lecture Notes in Computer Science, Vol. 2622. Springer Berlin Heidelberg, 153–169. DOI: http://dx.doi.org/10.1007/3-540-36579-6_12

[26] Li Li, Alexandre Bartel, Tegawendé François D Assise Bissyande, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. IccTA: detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE 2015)*.

[27] Steffen Lortz, Heiko Mantel, Artem Starostin, Timo Bähr, David Schneider, and Alexandra Weber. 2014. Cassandra: Towards a Certifying App Store for Android. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*. ACM, 93–104.

[28] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 2012. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. ACM, New York, NY, USA, 229–240. DOI: http://dx.doi.org/10.1145/2382196.2382223

[29] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. 2015. Composite constant propagation: Application to android inter-component communication analysis. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*.

[30] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. 2013. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *USENIX Security 2013*.

[31] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. 2014. A machine-learning approach for classifying and categorizing android sources and sinks. *2014 Network and Distributed System Security Symposium (NDSS)* (2014).

[32] Siegfried Rasthofer, Steven Arzt, Stephan Huber, Max Kohlhagen, Brian Pfretschner, Eric Bodden, and Philipp Richter. 2015. Droidsearch: A tool for scaling android app triage to real-world app stores. *Proceedings of the IEEE Technically Co-Sponsored Science and Information Conference* (2015).

[33] Siegfried Rasthofer, Steven Arzt, Enrico Lovat, and Eric Bodden. 2014. Droidforce: enforcing complex, data-centric, system-wide policies in android. In *Availability, Reliability and Security (ARES), 2014 Ninth International Conference on*. IEEE, 40–49.

[34] Siegfried Rasthofer, Steven Arzt, Stefan Triller, and Michael Pradel. 2017. Making Malory Behave Maliciously: Targeted Fuzzing of Android Execution Environments. In *Proceedings of the 39th International Conference on Software Engineering*. ACM. To appear.

[35] Johannes Späth, Lisa Nguyen, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *European Conference on Object-Oriented Programming (ECOOP)*.

[36] Ryan Stevens, Clint Gibler, Jon Crussell, Jeremy Erickson, and Hao Chen. 2012. Investigating user privacy in android ad libraries. In *Workshop on Mobile Security Technologies (MoST)*. 10.

[37] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '99)*. IBM Press, 13–. http://dl.acm.org/citation.cfm?id=781995.782008

[38] Raja Vallee-Rai and Laurie J. Hendren. 1998. Jimple: Simplifying Java Bytecode for Analyses and Transformations. (1998).

[39] Nicolas Viennot, Edward Garcia, and Jason Nieh. 2014. A Measurement Study of Google Play. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '14)*. ACM, New York, NY, USA, 221–233. DOI: http://dx.doi.org/10.1145/2591971.2592003

[40] IBM Watson. Watson libraries for analysis. (????). http://wala.sourceforge.net/wiki/index.php

[41] The Next Web. 2014. 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/. (2014). Accessed: 2017-01-13.

[42] R Winsniewski. 2012. Android–apktool: A tool for reverse engineering android apk files. (2012).

[43] Dacong Yan. 2014. *Program analyses for understanding the behavior and performance of traditional and mobile object-oriented software*. Ph.D. Dissertation. Ohio State University.

[44] Z. Yang and M. Yang. 2012. LeakMiner: Detect Information Leakage on Android with Static Taint Analysis. In *2012 Third World Congress on Software Engineering*. 101–104. DOI: http://dx.doi.org/10.1109/WCSE.2012.26

[45] Mu Zhang and Heng Yin. 2014. AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications. *Proceedings of the 21st Network and Distributed System Security (NDSS) Symposium* (2014).