Dynamic Taint-tracking: Directions for Future Research

Fabian Berner and Johannes Sametinger
Dept. of Business Informatics, LIT Secure and Correct Systems Lab,
Johannes Kepler University Linz, Austria

Keywords: Android, Information Disclosure, Taint-analysis, Taint-tracking.

Abstract: Detection of unauthorized disclosure of sensitive data is a dynamic research field. We can protect sensitive data on mobile devices through various commercial, open source and academic approaches. Taint-tracking systems represent one of the approaches to detect information disclosure attacks. In this paper, we give an overview of taint-tracking systems for Android. We discuss the systems and their shortcomings. The contribution of this paper is to provide an overview of Android taint-tracking systems, and to reveal directions for future research.The overview can serve as a basis for the selection of a taint-tracking system in specific situations.

1 INTRODUCTION

Mobile devices like smartphones, tablets and smartwatches have become ubiquitous in recent years. They have advantages and disadvantages. On the one hand, mobile Internet access allows to look up information, to check emails, to schedule meetings and appointments, or to navigate. Mobile devices also include cameras that allow us to take pictures at any time, and are used for many other purposes. On the other hand, due to the permanent connection to the Internet, we're available 24/7. Since the amount of sensitive data stored on mobile devices has increased, they have become worthwhile targets for attackers. Attacks can pursue a direct purpose like *espionage*, *spamming*, or provide *targeted advertising*. Stolen information can also be used by cybercriminals for other attacks like *social engineering*, *spoofing*, *phishing* or other *frauds*. In particular, the danger of frauds has increased recently because of commercial and payment services are now being also available on our mobile devices. Examples are e-commercial services like *Amazon* or *EBay* and financial services like *PayPal* or *Google Pay*.

A broad overview of Android security systems can be found for example in (Sufatrio et al., 2015; Xu et al., 2016; Tam et al., 2017). These ones as well as other similar studies do not focus on dynamic taint analysis but rather discuss various security system approaches for Android. Most papers dealing with novel taint-tracking systems also provide comparisons to distinguish their specific system from other ones, e.g., (Sun et al., 2016; You et al., 2017). This article is focused on taint-tracking systems for Android that can be used to detect app-based information disclosure attacks by monitoring information flows between a data source and data sink. The first part of this paper gives an overview of existing systems. This overview can serve as a basis for system selection. In the second part, we will evaluate taint-tracking systems, identify their shortcomings, and reveal possible future work.

2 DETECTION OF INFORMATION DISCLOSURE ATTACKS

Information disclosure means "unauthorized disclosure" of "sensitive data" (Shirey, 2007). To detect *information disclosure* attacks, we focus on *dynamic* taint-tracking techniques (or taint-analysis). In dynamic analysis systems, the object of investigation is executed and monitored, whereas a static analysis system analyzes the source code without execution. In general, both techniques are appropriate to detect information disclosure attacks. The main advantage of dynamic analysis is that apps can be tested in the same system, configuration and the same libraries as on the target system. Therefore, it is possible to detect attacks that need certain prerequisites. A *colluding apps* attack for example can only start its malicious function if all colluding apps are installed. In (Zheng et al., 2014), *Zheng et al.* discuss the prob-

Berner, F. and Sametinger, J.
Dynamic Taint-tracking: Directions for Future Research.
DOI: 10.5220/0008118502940305
In *Proceedings of the 16th International Joint Conference on e-Business and Telecommunications (ICETE 2019)*, pages 294-305
ISBN: 978-989-758-378-0

lems of static analysis in the context of apps written in Java. In particular, they see the missing ability to analyze *dynamically bound* code (e.g., with *reflection* or *generics*) as a major problem of static analysis approaches. *Wei et al.* argue in (Wei et al., 2014) that static analysis of Android apps is particularly challenging because the control flow of Android apps is event-based and therefore unpredictable. *Xia et al.* add that some of the code paths identified in static analysis "could never happen in real execution" (Xia et al., 2015). Other big challenges are *code obfuscation* and *code encryption*. Encrypted or obfuscated code cannot be analyzed by a static analysis system. For attackers, these are easy ways to hide a malicious code. Compared to static analysis, dynamic analysis appears to be more promising to detect information disclosure attacks so far. The main disadvantage of dynamic analysis is that only the executed program paths are analyzed. This leads to problems especially in short analysis processes with low path coverage. Nevertheless, it has to be mentioned that hybrid analysis, i.e., a combination of static and dynamic analysis, can boost the quality of security analysis, e.g., see (Graa et al., 2015; Rasthofer et al., 2016). Typically, static analysis is performed as an upstream analysis and the results are used as input for dynamic analysis. On the basis of this upstream analysis, the dynamic analysis can be used to run through specific execution paths of the program in a more targeted manner and thus potentially find more security gaps.

3 TAINT-TRACKING

Taint-tracking is a data flow analysis technique by marking specific data with a *taint tag* or *taint* for short. The taint itself is transparent for programs that use this data. Therefore, it can be imagined as a watermark. Tainted data originates from a *taint source* and leaves the system in a *taint sink*. Typically, taint source and the taint sink are system library methods. Taint-tracking means monitoring the data flow between taint source and taint sink. To evaluate the usage of the tainted data, the *taint propagation* logic monitors the program flow and detects whether the tainted data is processed or copied. If tainted data is copied, the copy is also marked with the same taint as the original data. Taint tags are stored in a taint tag storage. *Overtainting* describes the case when insensitive data is tainted and monitored. Overtainted data flowing to a taint sink leads to *false positives*. *Undertainting* describes the opposite, when sensitive data is not marked by the taint-tracking. Undertainting can lead to *false negatives*.

Different dynamic taint-tracking systems exist for Android. Most of them are based on *Taintdroid*, while some are based on Minemu, TaintART or TaintMan. *Taintdroid* is used to date by default as a standard for taint-tracking in Android, and it has been extended by other academic security systems. Taintdroid was implemented for the Dalvik Virtual Machine (VM) but has not been ported to the newer Android Runtime (ART). *TaintART* (Sun et al., 2016) is an unofficial successor of Taintdroid and is based on Android's ART. The authors of TaintART are planning to release an implementation as an open source. *Minemu* is a taint-tracking system integrated in an emulator (Bosman et al., 2011). Its latest version was published in 2011 and is therefore outdated. TaintMan (You et al., 2017) is also an unofficial successor of Taintdroid. It can be executed on the Dalvik VM as well as in ART. Subsequently, we will focus on Taintdroid, TaintART and TaintMan only.

3.1 Taintdroid

Taintdroid is the best known taint-tracking system for Android. It was published in 2010 by *William Enck et al.*, (Enck et al., 2010) who developed it within the scope of his PhD thesis under the title *Analysis Techniques for Mobile Operating System Security* (Enck, 2011). An enhancement to the Taintdroid concept was published in 2014 (Enck et al., 2014).

Taintdroid has been used by many security systems for Android, mostly in academia because of its advanced development and its availability as open source. In the design of any dynamic security analysis system, developers have to decide between performance and storage requirements. Generally speaking, it is possible to reduce the performance overhead of a security analysis system by using more storage capacity and vice versa. Taintdroid's tracking approach is optimized to minimize performance and storage overhead by limiting the amount of different kinds of data sources. The majority of taint-tracking systems uses a *shadow memory* or *taint map*. These are both being specific data structures that contain taint information (Enck, 2011, p. 58).

Taint Tag Storage. In Taintdroid, the *taint tag storage* is called *virtual taint map*. To store taints in the virtual taint map, Taintdroid uses a 32-bit vector for each data type provided by Dalvik (*method local variable*, *method arguments*, *class static fields*, *class instance fields* and *arrays* (Enck, 2011)). Each of the 32 bits stands for a specific taint tag type and thereby for a defined set of sensitive data. Taintdroid is therefore limited to 32 types of trackable data.

Taint Propagation Logic. Taintdroid uses four tracking techniques to monitor flows of (1) *variables*, (2) *method parameters*, (3) *files* and (4) *Inter Process Communication (IPC) messages* that contain sensitive data.

Discussion. Taintdroid is often cited in the context of taint-tracking and some of its limitations are discussed in different publications. The following provides a comprehensive collection of Taintdroid's limitations, indicating the need for further research:

- *32-bit vector:* Taintdroid is limited to 32 different kinds of trackable data (Enck et al., 2010).
- *Native code:* A recurring problem is the analysis of native code. Like most of the presented systems, Taintdroid is only able to analyze bytecode (Enck et al., 2010).
- *Covert channels:* Taintdroid is only able to track data over overt channels, e.g., IPC and Inter Component Communication (ICC)). (Sarwar et al., 2013).
- *Implicit data flows:* Implicit data flows, e.g., based on comparison, in which tainted data is only read but not (directly) copied, cannot be detected (Sarwar et al., 2013).
- *Overtainting:* The purpose of device identifiers (IMSI, MCC, MNC, MSIN) is to provide a unique reference number, which can be used to identify a specific device. Taintdroid is unable to distinguish between a mere identification and an information disclosure attack.
- *Late reporting:* Taintdroid reports to the user immediately after the tainted information has been sent to a taint sink. That means that the data has left the mobile device before the user has had any chance to react.
- *Version:* Taintdroid is based on an obsolete Android version.

The detection of malicious code can be prevented while Taintdroid's security analysis is running, i.e., by *sandbox-detection* and by *evasion techniques*. Sandbox-detection includes several techniques to detect whether execution is monitored by a security analysis system. Evasion techniques prevent detection, for example, by awaiting the end of security analysis. Attacks can also use weaknesses of Taintdroid's taint propagation mechanism. Such attacks are discussed by *Sarwar et al.* in (Sarwar et al., 2013). Finally, covert channels can bypass the official IPC mechanisms and therefore cannot be tracked by Taintdroid.

Security Systems based on Taintdroid. A variety of security systems is based on Taintdroid. The systems we focus on and present in this section, detect information disclosure attacks and help to protect sensitive data on Android devices. However, Taintdroid has also been used as basis for different security systems. For example, *AppFence* by *Hornyack et al.* helps to diminish the harm of app-based information disclosure attacks by using fake rather than real sensitive data.

MOSES, short for *MOde-of-uses SEparation in Smartphones*, presented by *Russello et al*, (Russello et al., 2012; Zhauniarovich et al., 2014) provides app isolation (*soft virtualization*) by policies that can be modified at runtime. MOSES is a program that places apps and the respective sensitive data in isolated profiles, e.g., *work*, *private* and *default* (Hornyack et al., 2011). MOSES uses Taintdroid taint propagation mechanism to monitor information flows between different profiles.

TreeDroid by *Dam et al.* is another policy-based security system that uses Taintdroid's data flow tracking technique (Dam et al., 2012). TreeDroid generates tree automata for a given app and provides fine-grained policy enforcement at runtime.

VetDroid by *Zhang et al.* analyzes app behavior based on a permission usage analysis (Zhang et al., 2013). After granting permissions in Android, it is transparent to the user how permissions are used by the app. VetDroid is able to "reconstruct permission use behavior" (Zhang et al., 2013) of a given app.

YAASE, short for *Yet Another Android Security Extension*, by *Russello et al.* focuses on information disclosure via network and colluding applications (Russello et al., 2011). YAASE labels and tracks *app-to-app* communication as well as *app-to-internet* communication. This labeling (taint) mechanism is based on a modified Taintdroid where the user can determine the data that should be examined and how it should be labeled.

AppsPlayground developed by *Rastogi, Chen and Enck* (Rastogi et al., 2013) is a modular and scalable dynamic analysis framework for testing Android applications. It is based on Taintdroid and additional dynamic detection techniques like system call monitoring. AppsPlayground automatically evaluates a given app by triggering several kinds of system events based on redundancy avoiding, a heuristic-based execution technique.

Only little information is available on the emulator-based security system *DroidBox* by *Lantz and Delosieres* (cf. (Lantz and Delosieres, 2015)). The open source system is available for Taintdroid 4.1.1, using the machine emulator Quick Emulator

(QEMU) (QEMU Project, 2017). At the beginning, a QEMU instance is started with a *Taintdroid* Android Virtual Device (AVD). Security analysis is done by the `droidbox.py` script, which collects data from different sources, e.g., network data and API file operations.

Graa et al. propose a *hybrid* security analysis approach based on Taintdroid (Graa et al., 2015). Using an upstream static analysis placed in the *Dalvik VM Verifier*, it allows to inspect the control flow of an app. The results of the static analysis are used as initial parameters for the dynamic analysis based on Taintdroid.

Andrubis is based on *Anubis*, a dynamic malware analysis sandbox for Windows applications (Weichselbaum et al., 2014; Lindorfer et al., 2014). Andrubis follows a hybrid analysis approach with an upstream static analysis for feature extraction, e.g., *AndroidManifest.xml*) and a dynamic behavior analysis at runtime. The dynamic analysis part is based on Taintdroid, method tracing and a modified Dalvik VM as a system analysis component. The analysis run is fully automated, for example with *Monkey* as user input generator.

Mobile-Sandbox provides a website, which apps can be uploaded to for a hybrid analysis (Spreitzenbarth et al., 2013; Spreitzenbarth et al., 2015) . For the security analysis, different existing security systems are integrated. Static analysis is, for example, done by tools like *Droidlyzer* and the Static Android Analysis Framework (SAAF). The results are used as input for the dynamic analysis tools like Taintdroid and DroidBox.

QuantDroid is an extension for Taintdroid by *Markmann, Mollus and Westhoff* (Mollus et al., 2014). QuantDroid is a dynamic information flow analysis that generates flow-graphs to quantify information flows between Android processes. The idea is to define thresholds for communication between specific processes. For example, no data exchange or only a specific amount of data is allowed over a specific communication channel.

Qian et al. suggest a system called *NDroid*, which extends Taintdroid to track information flows through Java Native Interface (JNI) based on a modified QEMU VM (Qian et al., 2014).

3.2 TaintART

TaintART by *Sun, Wei and Lui* was published in late 2016 (Sun et al., 2016). Since the Dalvik VM, which Taintdroid is based on, was replaced by ART in Android 5.0 (released in November 2014), Taintdroid has become outdated. The public release of TaintART's source code has been announced but is not available so far. The prototypical implementation was done for Android 6.0 (Sun et al., 2016), which uses ART as runtime environment for apps. The *TaintART compiler* integrates the taint logic into the compiled application. At runtime, *TaintART runtime* executes the compiled native code and tracks tainted data. TaintART distinguishes between four levels of information disclosure, from low to high security needs (Sun et al., 2016).

Taint Tag Storage. To speed up taint-tracking, TaintART uses a CPU register as the fastest storage on a computer for taint tag storage. The prototypical implementation is built on a 32-bit ARM processor as target architecture and can therefore save 32 taint tags. To avoid the limitation of registers, TaintART is also able to temporarily store additional taint tags in the device's memory.

Taint Propagation Logic. TaintART provides three taint propagation logic methods: (1) *basic taint propagation*, (2) *taint propagation via methods calls*, and (3) *propagation between apps through Binder IPC*. For the basic taint propagation, the ART compiler generates a Control Flow Graph (CFG), which is used by the TaintART compiler to instrument the source code for variable-level taint-tracking. In case a method is called, TaintART uses an invocation taint propagation to track tainted data in method parameters. The third part is a message-level propagation for Android's Binder IPC.

Discussion. Compared with Taintdroid, the advantage of TaintART is its compatibility with ART which is the new runtime environment in Android. Therefore, TaintART is an interesting alternative to Taintdroid: on the one hand, it is partly similar to Taintdroid and, on the other hand, it improves the concept of taint-tracking. Compared to Taintdroid, one of the improvements is the possible integration of *NDroid* for analyzing native code.

Attacks against TaintART. To the best of the authors' knowledge, there is no specific attack vector published yet that targets TaintART. Similar to Taintdroid, TaintART's analysis can also be circumvented by collaborating applications that use covert channels to bypass TaintART's analysis mechanism.

Since TaintART's source code has not been released yet as of this writing, extensions or systems based on TaintART have not been published yet.

3.3 TaintMan

Another unofficial successor of Taintdroid is *TaintMan* by *You et al.*, which integrates the taint-propagation logic in the apps and libraries to analyze (You et al., 2017). Therefore, TaintMan uses a desktop computer program to statically instrument the bytecode. First, the app to be analyzed is unpacked, then the bytecode is instrumentated, the app's entry point is changed. Finally, the app is repacked. In Android, apps can check their own integrity based on digital signatures. In (You et al., 2017), the authors discuss the possibility to bypass Androids system library. The tool is also able to download libraries from an Android device, instrument them and upload the libraries back to the device. Once uploaded and instrumented, the system libraries do not replace the original ones and are stored in a separate directory.

The release of TaintMan has not been announced, and the program has not been available yet as of this writing. The prototypical implementation seems to be developed for Android 4.0 (which uses Dalvik VM) and 5.0 (which uses ART) (You et al., 2017).

Taint Tag Storage. Similarly to Taintdroid, TaintMan uses a 32-bit vector for each variable to encode the taint tag. For this reason, the amount of different taint sources is also limited to 32. Instead of using a taint map for storing the taint tags, like in Taintdroid, TaintMan stores all taint tags for local variables, parameters, return values and exceptions on an internal stack. Taint tags for class variables are stored in shadow fields in the corresponding classes. As in Taintdroid, only one taint tag per array is used to minimize the storage needs of TaintMan.

Taint Propagation Logic. TaintMan imbeds its taint-tracking commands in the apps or libraries bytecode. Therefore, its analysis capabilities are limited to instrumented apps and libraries. This is a main difference in comparison to other Dynamic Taint Analysis (DTA) systems discussed, which integrate the taint-tracking logic in the mobile Operating System (OS). To save resources, TaintMan tries to instrument as few taint-tracking instructions in the app's bytecode as possible. A detailed overview of TaintMan's taint propagation logic commands, which can be instrumented in apps and libraries, can be found in (You et al., 2017).

Discussion. TaintMan can execute taint-analysis on non-rooted Android devices, which is a huge advantage for its usability as well as for the quality of the analysis. The quality of analysis has been improved as the execution on a real device is more lifelike and much harder to detect for an attacker.

Since TaintMan has not been released yet as of this writing, no independent evaluations of TaintMan are available. In (You et al., 2017) the authors attest TaintMan high effectiveness regarding detection of information disclosure attacks with *"acceptable"* (You et al., 2017) performance (42,3 % without and 28,9 % overhead with code optimizations (You et al., 2017)) and storage overheads (instrumented code is about 23 % larger (You et al., 2017)). Such overheads are acceptable in case of software tests, but they are not suitable for everyday use. Increased performance overhead leads to decreased battery lifetime.

A comparison of TaintMan with Taintdroid or TaintArt is difficult:

- TaintMan uses the newer ART runtime environment, whereas Taintdroid uses the outdated Dalvik VM. In most cases, execution of apps is much faster in ART than in Dalvik VM.
- Both Taintdroid and TaintArt are not publicly available and, therefore, cannot be tested or evaluated.

Attacks against TaintMan. Specific attack vectors that target TaintMan have not been reported so far. Similarly to Taintdroid and TaintART, TaintMan analysis can also be circumvented by collaborating applications that use covert channels to bypass TaintMan's analysis mechanism.

Without the availability of the source code, only theoretical concepts can be built. Extensions or systems based on TaintMan have not been published yet.

4 COMPARISON

Without the source code of TaintART and TaintMan, our knowledge about both systems is based on the information given in (Sun et al., 2016) and (You et al., 2017). Taintdroid, TaintART and TaintMan must be compared with caution. Taintdroid is well known and has been described earlier. Its functions and limitations have been discussed in many papers. In contrast, both TaintART and TaintMan have not received that much attention yet. We expect TaintART and TaintMan to display similar limitations. It remains to be seen whether TaintART or TaintMan can close the gap left by Taintdroid.

The subsequent system comparison shows the differences among these systems and can serve as a basis for system selection in a specific scenario. In the tables below a check mark (✓) shows that a property

is fulfilled and a dash (-) shows that it is not fulfilled by the respective system. If the analysis technology cannot be clearly classified by the given scheme, the analysis techniques are marked with a question mark (?). Tendencies are marked with a check mark or dash in front of the question mark. Tendencies occur when a property is incompletely (✓(?)) or even rudimentary fulfilled (- (?)) by the system. In case the property is not applicable, the abbreviation *n/a* is used.

All information and system characteristics are taken from the papers cited above. Most systems are not publicly available and thus can not be investigated any further. Another limitation is the classification scheme: Some systems have the characteristics of more than one classification class. In these cases, the system is attributed to the closest class.

4.1 General Information

We can roughly distinguish among the following three types of users:

- *Average User.* User without any special knowledge of computer security.
- *Administrator or Pen Tester.* User with special knowledge in computer security who has to test the security of an app or has to adjust the configuration of an OS or app for security reasons.
- *App Developer.* User who wants to validate and test an app.

Most systems do not specify their target group. In this case, we determine it as *normal users* for every live analysis system in which the user has to decide on the security relevant issues. All systems that run ahead of the app usage can also be used by administrators or pen testers. This also applies to systems that are configured once and run without user interaction.

Table 1 provides basic information about the systems: the OS and its version (or version range), the existing security software the system is based on as well as the probable target group. We have marked the version with *n/a* when we were not able to find any information about it.

Table 1: General Information and Assumed Target Group.

Name	Base	Year	Version	avg	admin	dev
TaintDroid (TD)	-	2011	2.X - 4.3	-	(?)	(?)
TaintART	-	2016	6.0	-	(?)	(?)
TaintMan	SB	2017	4.0 - 5.0	-	(?)	(?)
Andrubis	TD, A	2012	n/a	-	✓	✓
AppFence	TD	2011	2.1	(?)	-	-
AppsPlayground	TD	2013	n/a	-	✓	✓
DroidBox (DB)	TD	n/a	2.3 - 4.1.1	-	✓	(?)
Graa et al.	TD	2015	n/a	-	(?)	(?)
Mobile Sandbox	TD/DB	2013	n/a - 4.1.X	-	✓	(?)
MOSES Droid	TD	2012	2.3.4_r1	-	✓	(?)
Ndroid	TD, DS, D	2014	n/a - 4.1.1_r6	-	✓	(?)
TreeDroid	TD, D2J	2012	2.3	-	✓	(?)
QuantDroid	TD	2012	2.3 - n/a	-	✓	✓
VetDroid	TD	2013	2.3	-	✓	(?)
YAASE	TD	2011	n/a	-	✓	(?)

Legend: avg ... average user, admin ... administrator/pen tester, dev ... app developer; TD ... TaintDroid, A .. Anubis, D ... Darm 1), D2J ... dex2jar 2), DB ... DroidBox, DS ... DroidScope, SB ... Smali/Baksmali 3)
1) *https://github.com/jbremer/darm*, 2) *https://sourceforge.net/projects/dex2jar/*, 3) *https://github.com/JesusFreke/smali/wiki*

4.2 Analysis

All systems use a dynamic analysis approach, which is enhanced by a few systems to a hybrid analysis. They are integrated in the mobile OS, and only for TaintART the mobile OS has to be rooted additionally. The property device (*emulator* or *real device*) deserves special attention as well as the possibility for *iterative security analysis*. Systems designed to run on real devices cannot detect every malicious behavior. Note that every false-negative ends in an information disclosure of sensitive data. Even worse, the warning message in Taintdroid is not generated until the sensitive data we want to protect has already left the device. No doubt, a solution can be found to counteract this weakness. It occurs only because Taintdroid is of academic origin and has to be developed further.

Since mobile devices have limited resources, the location where the security analysis is executed is another interesting characteristic. Basically, there exist *host-based* security analysis approaches that are executed on the mobile device itself, and *distributed* approaches that help to minimize the needed resources. All systems except for QuantDroid and Mobile Sandbox are host-based. If the security system is executed on an emulated device, the difference between both characteristics can mostly be neglected.

The ability to run iterative tests and the comparability of two or more test runs have yet to be addressed. All shown systems are based on monitoring and offer no features for comparison because monitoring is a process without a clearly defined start or end of analysis. In contrast, systems that run in an emulation environment normally use a scanning approach, which is started and stopped for an analysis run. None of the publications on the systems compared above have mentioned the possibility of comparing the results of analysis runs. None of them are able to analyze native code. However, *You et al.* have disscussed the possible integration of *NDroid* to extend its analysis capabilities to native code (You et al., 2017).

Table 2 summarizes information about the secu-

Table 2: Security Analysis.

	Static	Dynamic	Integrated in OS	Rooted OS	Device[1]	Host-based	Distributed	Iterative analysis?[2]	A single app	Two or more apps	Complete system	Native Code
TaintDroid	-	✓	✓	-	r	✓	-	m	✓	✓	✓	-
TaintART	-	✓	✓	✓	r	✓	-	m	✓	✓	✓	-
TaintMan	✓	✓	-	-	r	✓	-	m	✓	✓	-	-
Andrubis	✓	✓	✓	-	e	✓	-	s	✓	✓	-	-
AppFence	-	✓	✓	n/a	r	✓	-	m	✓	✓	✓	-
AppsPlayground	✓	✓	✓	-	e	✓	-	s	✓	✓	-	-
DroidBox	-	✓	✓	-	e	✓	-	s	✓	✓	-	-
Graa et al.	✓	✓	✓	-	r	✓	-	m	✓	✓	✓	-
Mobile Sandbox	✓	✓	✓	-	e	✓	(?)	s	✓	✓	-	-
MOSES Droid	-	✓	✓	n/a	r	✓	-	m	✓	✓	✓	-
Ndroid	-	✓	✓	-	e	✓	-	s	✓	✓	-	-
TreeDroid	-	✓	✓	n/a	r	✓	-	m	✓	✓	✓	-
QuantDroid	-	✓	✓	n/a	r/e	✓	✓	s	✓	✓	✓	-
VetDroid	-	✓	✓	n/a	r/e	✓	-	m/s	✓	✓	-	-
YAASE	-	✓	✓	n/a	r	✓	-	m	✓	✓	✓	-

[1] r ... real device; e ... emulator-based
[2] s ... scanner; m ... monitor

rity analysis approach. All taint-tracking systems mentioned above can be used to evaluate the security of a single app. At least theoretically, they should also be able to evaluate two or more apps, for example, in order to detect colluding apps. Since most of the systems are not publicly available, it is impossible to check this out. The system's ability to secure the whole device is questionable: Taintdroid and TaintART were developed to monitor information disclosure attacks on the application level. Neither Taintdroid as the basis of the discussed systems nor TaintART monitor the OS itself. The second part of Table 2 shows the analysis range of the security systems. Analysis range means the scope analyzed by the system: (1) a single app, (2) two or more apps, and (3) the complete OS with all running apps and 3rd party libraries. The analysis range is important because attack vectors can stretch over more than one app, cf. colluding apps. Systems that are limited to one app are not able to detect such attack vectors. The category *complete system* means that all apps that run together in one OS instance can be analyzed simultaneously. However, the term *complete system* does not refer to the OS itself.

Taintdroid and TaintART focus solely on dynamic taint-tracking for the detection of information disclosure attacks. Most of the mentioned systems combine the dynamic taint-tracking approach with some additional analysis techniques. TaintART and Taintdroid are integrated in the Android's app runtime environment and can therefore use analysis data from the OS. This also affects other security systems based on Taintdroid. Some of the systems additionally extract analysis data from the apps or rather the source code after decompiling the app. None of the systems uses data sources from outside the analyzed device. Analyzing data from outside the analyzed device can be important to security analysis, especially for detecting information disclosure attacks.

The following classification of static and dynamic security systems is adopted from (Neuer et al., 2014).

Static security analysis systems:

- *Extraction of Metadata.* "Tools extract information from an application's manifest and provide information about requested permissions, activities, services and registered broadcast receivers. Meta information is often used during later dynamic analysis in order to trigger an application's functionality."
- *Weaving.* "Tools rewrite bytecode of applications using a bytecode weaving technique. This allows them, for instance, to insert tracing functionality into an existing application."
- *Decompiler.* "Tools implement a Dalvik bytecode decompiler or disassembler."

Dynamic security analysis systems:

- *Taint-tracking.* "Taint-tracking tools are often used in dynamic analysis frameworks to implement system-wide dynamic taint propagation in order to detect potential misuse of users' private information."
- *Virtual Machine Introspection (VMI).* "VMI-based frameworks ... intercept events that occur within the emulated environment. Dalvik VMI-based systems monitor the execution of Android APIs through modifications in the Dalvik VM. Qemu VMI-based systems are implemented on the emulator level to enable the analysis of native code. However, emulators are prone to emulator evasion."
- *System Call Monitoring.* "Frameworks collect an overview of executed system calls, by using, for instance, VMI, strace or a kernel module. This enables (partial) tracing of native code."
- *Method Tracing.* "Frameworks trace Java method invocations of an app in the Dalvik VM."

We distinguish the following possible sources of analysis data:

- *App or Source Code:* About half of the systems gain information from the analyzed app, the compiled bytecode or the source code.
- *Meta Data:* Also about half of the systems evaluate app meta data like the manifest file of Android apps.
- *Mobile OS:* The OS generates information that can be used for security analysis like systems log files. Furthermore, altered Application Programming Interfaces (APIs) etc. can be used to generate additional analysis data. All systems dis-

cussed above use information from the mobile OS in some form.

- *Agents and Network:* None of the systems discussed above uses agents or other network analysis techniques outside the mobile OS.

Table 3 summarizes security analysis approaches used by the different taint-tracking systems, which implement different analysis methods for *static* and *dynamic* analysis.

Table 3: Static and dynamic analysis methods.

	Static Analysis				Dynamic Analysis				Analysis Data Source					
	Metadata Extraction	Weaving	Taint-tracking	Decompiler	Taint-tracking	VM Introspection	System Call Monitor	Method tracing	Adjustable data collection	App or source code	Metadata	OS	Agents	Network
TaintDroid	-	-	-	-	✓	-	-	-	- (?)	-	-	✓	-	-
TaintART	-	-	-	-	✓	-	-	-	✓	✓	-	✓	-	-
TaintMan	-	✓	-	✓	✓	✓	-	-	- (?)	✓	-	-	-	-
Andrubis	✓	-	-	✓	✓	✓	✓	✓	-	✓	✓	✓	-	-
AppFence	-	-	-	-	✓	-	✓(?)	-	-	-	✓	✓	-	-
AppsPlayground	-	-		-	✓	n/a	✓	n/a	-	✓	n/a	✓	-	-
DroidBox	-	✓(?)	-	-	✓	-	-	-	-	-	✓(?)	✓	-	-
Graa et al.	✓	n/a	n/a	n/a	✓	-	-	-	-	✓	n/a	✓	-	-
Mobile Sandbox	✓	n/a	-	✓	✓	n/a	✓(?)	✓(?)	-	✓	✓	✓	-	-
MOSES Droid	-	-	-	-	✓	-	n/a	n/a	-	-	✓	✓	-	-
Ndroid	-	-	-	-	✓	n/a	✓(?)	✓(?)	-	-	-	✓	-	-
TreeDroid	-	-	-	-	✓	-	- (?)	- (?)	-	n/a	n/a	✓	-	-
QuantDroid	-	-	-	-	✓	-	✓	✓	-	-	✓(?)	✓	-	-
VetDroid	-	-	-	-	✓	- (?)	n/a	n/a	-	-	n/a	✓	-	-
YAASE	✓	-	-	-	✓	-	✓	-	-	-	✓	✓	-	-

5 POTENTIAL RESEARCH DIRECTIONS

All security systems described above share a subset of shortcomings. We will discuss these shortcomings along with directions for possible research.

5.1 Detection Rates

No security system is able to detect all possible information disclosure attacks. This is in the nature of things, but it proves that analysis at runtime on a mobile device with real sensitive data is problematic. Every false-negative on a real mobile device that stores real sensitive data leads to a real information disclosure, which is irreversible. On real mobile devices, it is also problematic to execute an upstream analysis before executing the app because it slows down the start of the app. Another point is the analysis of apps that use native code functions or libraries. Most of the evaluated security systems are not able to analyze native code.

Potential Research. To increase detection rates (true-positives and true-negatives) and, thus, to decrease false alarms (false-positives), more security relevant analysis data is needed. Ideally, the analysis data should be enlarged by additional data sources that are independent from the existing ones. Independent here means that two analysis data sources should not depend on each other. For example, an information disclosure attack can be detected by the analysis of information flow among apps and by network traffic analysis. The analysis data collection of existing systems is often limited ...

- ... to one app at a time: a security system is not able to detect attack vectors like colluding applications, confused deputy or covert channels because the relevant analysis data cannot be collected.
- ... to the external borders of a mobile device: security systems are not able to collect analysis data beyond the device context. Especially for information disclosure attack detection, information flows in both directions – to and from a mobile device to other devices – seem to be an important source of analysis data.
- ... by time and resources: dynamic analysis on a mobile device suffers because of the strong limitations of the device resources. If security is evaluated by an upstream analysis system, it is limited by time in order to speed up the app installation or loading time.

The analysis data collection mechanism of dynamic analysis systems, which are executed live on a real mobile device, is often intentionally limited for performance reasons. For example *Taintdroid* supports only 32 different kinds of taint-sources for that matter.

5.2 Limited System Resources

Security analysis at runtime on a mobile device costs valuable resources. Even if devices become more powerful, we may not forget that Android penetrates also into other areas. *Android Wear* powered devices like smart watches are just one example. This shortcoming becomes evident only on security systems running on the mobile device directly.

Potential Research. There are three ways to decrease the performance overhead of mobile device security systems:

- Security analysis approaches can be optimized in favor of better performance. The problem is that the optimization potential is limited because it directly depends on the amount of analysis data. As discussed above, using additional data sources will lead to an additional slowdown of the security analysis.

- Performance of the mobile device can be increased by moving security analysis to an external system. Security systems like *Paranoid Android* (Portokalidis et al., 2010) and *Mobile-Sandbox* (Spreitzenbarth et al., 2015) are already using this approach. However, this leads to a further limitation because the next bottleneck on mobile devices is caused by network connectivity and performance. Additionally, trust is a big issue if security analysis does not run on the mobile device itself.
- Security analysis can be executed before the application is used on a mobile device. In terms of performance, this is the best solution because it generates no performance overhead on the mobile device. The problem of this approach is – of course – that the behavior of the app is not monitored at runtime. Eventually, this limitation could be mitigated by rerunning security analysis over the time of use.

5.3 User Interface Complexity

A runtime security analysis either has to be fully automated or the user has to react manually to identify potential security risks. Without expert knowledge in computer security, however, the average user may react incorrectly to the security systems warning. Another – maybe worse – reaction to messages of the security system could be deactivation of the detection mechanism in case the user is overwhelmed by the amount and complexity of security warnings.

Potential Research. The complexity of the existing security systems for average users is theoretically easy to reduce by transferring the responsibility from the average user to an expert. However, this responsibility transfer is accompanied by rising costs for the company, which originally did not intend to put the security in the hands of their employees. Nevertheless, it is irresponsible to force an average user to take security-relevant decisions.

5.4 Analysis Abstraction Level

Most of the security systems use a fully automated high level analysis or a manual low level analysis. A good balance between these two analysis approaches is rare. Especially the lack of customizability of the data collection mechanisms and security analysis is the major disadvantage of most existing systems.

Potential Research. There are reasons for low-level and high-level approaches; however, their focus is different: fully automated security analysis systems are designed to minimize the analysis effort for the user, whereas manually executed analysis, which is a high effort for the user, has other advantages like the ability to analyze specific parts – say system calls – of an application. A good balance between these two approaches can mitigate various disadvantages. Additionally, it is be beneficial if the security system user can choose the abstraction level on basis of the given app, the security objectives and a given suspicion as discussed below. To make the analysis level adaptable, future systems should be able to provide analysis methods of different analysis levels. A modular architecture, which can be extended with additional analysis methods, can satisfy this need.

The potential reseach directions of 5.3 and 5.4 go hand in hand. Systems that aim end users often try to automate the complete analysis process. In doing so, however, the systems are limited to detecting specific attacks (with signature-based analysis) or detecting abnormal behavior (with anomaly-based analysis). Abnormal behavior does not necessarily indicate that the system is under attack. Otherwise, ”normal” behavior does not necessarily indicate that the system is free of ongoing attacks.

5.5 Detection of Collaborating Apps

Taint-tracking systems can be used to detect information flows between apps. However, they are only able to analyze information flows that use overt channels like IPC.

Potential Research. All security analysis approaches which are limited to the analysis of one app are inappropriate. Future systems should allow, to install and execute more than one app at a time and to monitor possible information flows between these apps.

5.6 Security Analysis Rerun

Because of the short release cycles of Android apps as well as of Android itself, an automated rerun of security analysis and a subsequent comparison of analysis results seem to be an important function of a security system. In order to fully exhaust the potential of analysis reruns, *degrees of freedom* as well as *variations* should be considered. Variation arises through even minimal divergences in automatic security analysis reruns. Degrees of freedom instead are desirable because they enable the user to repeat a security analysis run with slightly different variables, if necessary.

Such variables can, for example, be a newer version of an app or a new version of a mobile device OS.

Potential Research. The inability of a system to rerun security analyzes is a major limitation of existing systems. However, this issue occurs only on upstream security analysis approaches, which have a defined start and end. For security analysis approaches which are not executed at runtime on mobile devices, it makes sense to enable the system to record an executed security test and to replay it automatically to rerun the security test for a given application. This can decrease the amount of work for testing because the responsible employee does not have to repeat the tests manually. Linked to an automated security test rerun functionality, other useful features should be discussed:

- An automatically executed test replay is particularly useful when certain degrees of freedom are possible. Examples for reasonable degrees of freedom are: app version, Android version as well as a variety of apps which are executed simultaneously.
- A second useful feature is the possibility to replay recorded security analysis runs in parts. Thereby, a rerun can be speeded up in case only some specific parts have to be analyzed, for example, when only a specific part of the original test run leads to an information disclosure attack.
- To save resources, the automatically executed test rerun can be speeded up by shortening pauses between user interactions and system events.
- A security system that provides automated test reruns does not have to be executed locally: at least the replays can be executed in a cloud or web service. In combination with the additional comparing function, the security analysis can be further automated: after an executed test rerun, the results of the original test run and of the actual one can be compared. If the security level had changed, that could also be reported to the user.

5.7 Granularity of Reaction

Most systems adopt the *all-or-nothing* principle. It means that possible reactions are not fine-grained enough. An example is *AppFence* that disconnects all internet connections as soon as a possible information disclosure is detected (Russello et al., 2011). In this case, it would actually be sufficient to interrupt only the connection through which the information has been leaked.

Potential Research. The security systems discussed above neither provide automated reactions nor suggest reactions to the user. For systems, no automated reaction system is needed when the system does not run on mobile devices with real sensitive data.

5.8 Suspicion-based Analysis

This shortcoming is closely connected with restrictions due to the predetermined analysis abstraction level. A fully automated analysis or a manual analysis system are not appropriate to analyze an application for a specific suspicion. A security analysis approach is needed that can be used to analyze the behavior of an app to confirm or rule out the suspicion.

Potential Research. A suspicion-based analysis can not be done with taint-tracking systems or most of the other security systems discussed previously. That means that these systems are not able to analyze a specific part of an application that seems suspicious for whatever reason. To analyze a specific suspicion, the analysis data collection and the analysis itself have to be adaptable; thus, a fully automated test run is not appropriate. For suspicion-based analysis, the data collection phase needs to be manually controlled by the user. Therefore, analysis systems which run in parallel are neither appropriate. In order to prevent misunderstandings: the inappropriate ability of automatic analysis systems to perform suspicion-based analysis should not be confused with automated replays, as discussed before. Automated replays can also be used in a suspicion-based analysis, for example to test a suspicious function of an app under different conditions.

Table 4 shows the mapping between shortcomings and taint-tracking systems. Stars (*) indicate a short-

Table 4: Summary of Shortcomings.

	Detection Rates	Limited system resources	User Interface complexity	Analysis abstraction level	Detection of collaborating apps	Security analysis rerun	Reaction granularity	Suspicion-based analysis
TaintDroid	*	* *	*	*	*	*	-	*
TaintART	*	*	*	*	*	*	-	*
TaintMan	*	*	*	*	*	*	-	*
Andrubis	*	-	-	*	*	*	-	*
AppFence	*	*	*	*	*	*	*	*
AppsPlayground	*	-	-	*	*	*	-	*
DroidBox	*	*	*	*	*	*	-	*
Graa et al.	*	*	*	*	*	*	-	*
Mobile Sandbox	*	-	-	*	*	*	-	*
MOSES Droid	*	*	*	*	*	*	*	*
Ndroid	*	-	-	*	-	*	*	*
TreeDroid	*	-	-	*	*	*	-	*
QuantDroid	*	*	*	*	*	*	-	*
vetDroid	*	-	-	*	*	*	-	*
YAASE	*	*	*	*	*	*	-	*

Legend: * ... shortcoming applies; - ... shortcoming does not apply

coming for a specific system. Dashes (-) are used when a shortcoming has not been proved to be true.

To address some of the above shortcomings, we propose a changed security analysis concept, that is based on dynamic taint-tracking: instead of a live analysis we propose an (1) upstream analysis approach, which is executed by a security expert before the the app is used in a productive way. The analysis is executed in a (2) virtualized environment, which allows an extended analysis data collection mechanism. Existing systems, that are executed on real mobile devices are limited by performance reasons. Since the virtualized device is not used in a productive way, it is possible to use fake data instead of real sensitive data. Thereby the analysis puts no real sensitive data on risk. To extend the system's architecture, the system needs to be built on an (3) extendable and open architecture. Our future work will target this security analysis concept.

6 SUMMARY

In this paper, advantages and disadvantages of static and dynamic security analyzes have been compared. Because of the limitations of static analysis, we have focused on dynamic analysis systems for detecting app-based information disclosure attacks. Android as the given target platform has narrowed the evaluated systems for analyzing Android apps.

To detect information disclosure attacks, taint-tracking seems to be a promising approach. At the time of this writing, neither the source code of TaintART nor that of TaintMan has been released, Taintdroid and its unofficial successors remain the main available taint-tracking approaches for analyzing Android apps. Some of the discussed systems seem to be outdated; however, since every system has its specific purpose, we believe it is necessary to review the research done on the topic and different approaches for improving taint-tracking. The evolved shortcomings show that there is a need for further research and new security systems.

Although all discussed shortcomings are important, we are convinced that particularly the replay of security analysis runs with the ability to compare the security analysis results are an important feature for future taint-tracking and other security systems. For all security systems that follow a scanner approach, the possibility of automated security analysis reruns with result comparison seems the only appropriate way given the amount of apps used, the short release cycles of apps and mobile device OS as well as the heterogeneity of mobile computing caused by the amount of mobile device manufactures, mobile device OS and versions.

REFERENCES

Bosman, E., Slowinska, A., and Bos, H. (2011). Minemu: The world's fastest taint tracker. In Hutchison, D. e. a., editor, *Recent Advances in Intrusion Detection*, volume 6961 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin Heidelberg.

Dam, M., Le Guernic, G., and Lundblad, A. (2012). Treedroid: A tree automaton based approach to enforcing data processing policies. In *Proceeding CCS '12 Proceedings of the 2012 ACM conference on Computer and communications security*, page 894.

Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. (2010). Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceeding OSDI'10 Proceedings of the 9th USENIX conference on Operating systems design and implementation*.

Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. (2014). Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems*, 32(2):1–29.

Enck, W. H. (May 2011). *Analysis Techniques for Mobile Operating System Security*. PhD thesis, Pennsylvania State University.

Graa, M., Cuppens-Boulahia, N., Cuppens, F., and Cavalli, A. (2015). Detection of illegal control flow in android system: Protecting private data used by smartphone apps. In Cuppens, F., Garcia-Alfaro, J., Zincir Heywood, N., and Fong, P. W. L., editors, *Foundations and Practice of Security*, volume 8930 of *Lecture Notes in Computer Science*, pages 337–346. Springer International Publishing.

Hornyack, P., Han, S., Jung, J., Schechter, S., and Wetherall, D. (2011). These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In Chen, Y., Danezis, G., and Shmatikov, V., editors, *Proceedings of the 18th ACM conference on Computer and communications security*, page 639.

Lantz, P. and Delosieres, L. (2015). Droidbox - android application sandbox.

Lindorfer, M., Neugschwandtner, M., Weichselbaum, L., Fratantonio, Y., Victor van der Veen, and Platzer, C. (2014). Andrubis - 1,000,000 apps later: A view on current android malware behaviors. In *Proceedings of the the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*.

Mollus, K., Westhoff, D., and Markmann, T. (2014). Curtailing privilege escalation attacks over asynchronous channels on android. In *14th International Conference on Innovations for Community Services (I4CS)*, pages 87–94.

Neuer, S., van der Veen, V., Lindorfer, M., Huber, M., Merzdovnik, G., Mulazzani, M., and Weippl, E.

(2014). Enter sandbox: Android sandbox comparison. In Koved, L., Singh, K., Chen, H., and Just, M., editors, *Proceedings of the Third Workshop on Mobile Security Technologies (MoST) 2014*.

Portokalidis, G., Homburg, P., Anagnostakis, K., and Bos, H. (2010). Paranoid android: versatile protection for smartphones. In Gates, C., Franz, M., and McDermott, J., editors, *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC '10)*, page 347.

QEMU Project (2017). Documentation/networking.

Qian, C., Luo, X., Shao, Y., and Chan, A. T. (2014). On tracking information flows through jni in android applications. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 180–191. IEEE.

Rasthofer, S., Arzt, S., Miltenberger, M., and Bodden, E. (February 21-24, 2016). Harvesting runtime values in android applications that feature anti-analysis techniques. In Capkun, S., editor, *Proceedings 2016 Network and Distributed System Security Symposium*. Internet Society.

Rastogi, V., Chen, Y., and Enck, W. (2013). Appsplayground: automatic security analysis of smartphone applications. In Bertino, E., Sandhu, R., Bauer, L., and Park, J., editors, *Proceedings of the third ACM conference on Data and application security and privacy*, page 209.

Russello, G., Conti, M., Crispo, B., and Fernandes, E. (2012). Moses: Supporting operation modes on smartphones. In *Proceedings of the 17th ACM symposium on Access Control Models and Technologies - SACMAT '12*, page 3. ACM Press.

Russello, G., Crispo, B., Fernandes, E., and Zhauniarovich, Y. (2011). Yaase: Yet another android security extension. In *2011 IEEE Third Int'l Conference on Privacy, Security, Risk and Trust (PASSAT) / 2011 IEEE Third Int'l Conference on Social Computing (SocialCom)*, pages 1033–1040.

Sarwar, G., Mehani, O., Boreli, R., and Kaafar, M. A. (2013). On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In Samarati, P., editor, *SECRYPT 2013, 10th International Conference on Security and Cryptography*. SciTePress.

Shirey, R. (2007). Rfc 4949: Internet security glossary, version 2.

Spreitzenbarth, M., Freiling, F., Echtler, F., Schreck, T., and Hoffmann, J. (2013). Mobile-sandbox: having a deeper look into android applications. In Shin, S. Y. and Maldonado, J. C., editors, *the 28th Annual ACM Symposium*, page 1808.

Spreitzenbarth, M., Schreck, T., Echtler, F., Arp, D., and Hoffmann, J. (2015). Mobile-sandbox: combining static and dynamic analysis with machine-learning techniques: Combining static and dynamic analysis with machine-learning techniques. *International Journal of Information Security*, 14(2):141–153.

Sufatrio, Tan, D. J. J., Chua, T.-W., and Thing, V. L. L. (2015). Securing android: A survey, taxonomy, and challenges. *ACM Computing Surveys*, 47(4):1–45.

Sun, M., Wei, T., and Lui, J. C. (2016). Taintart: A practical multi-level information-flow tracking system for android runtime. In Katzenbeisser, S. and Weippl, E., editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 331–342. Association for Computing Machinery.

Tam, K., Feizollah, A., Anuar, N. B., Salleh, R., and Cavallaro, L. (2017). The evolution of android malware and android analysis techniques. *ACM Computing Surveys*, 49(4):1–41.

Wei, F., Roy, S., Ou, X., and Robby (2014). Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In Ahn, G.-J., editor, *Proceedings of the 21st ACM Conference on Computer and Communications Security*, pages 1329–1341. ACM.

Weichselbaum, L., Neugschwandter, M., Lindorfer, M., Fratantonio, Y., van der Veen, V., and Platzer, C. (2014). Andrubis: Android malware under the magnifying glass.

Xia, M., Gong, L., Lyu, Y., Qi, Z., and Liu, X. (2015). Effective real-time android application auditing. In *2015 IEEE Symposium on Security and Privacy (SP)*, pages 899–914. IEEE.

Xu, M., Qian, C., Lee, S., Kim, T., Song, C., Ji, Y., Shih, M.-W., Lu, K., Zheng, C., Duan, R., Jang, Y., and Lee, B. (2016). Toward engineering a secure android ecosystem. *ACM Computing Surveys*, 49(2):1–47.

You, W., Liang, B., Shi, W., Wang, P., and Zhang, X. (2017). Taintman: An art-compatible dynamic taint analysis framework on unmodified and non-rooted android devices. *IEEE Transactions on Dependable and Secure Computing*, page 1.

Zhang, Y., Yang, M., Xu, B., Yang, Z., Gu, G., Ning, P., Wang, X. S., and Zang, B. (2013). Vetting undesirable behaviors in android apps with permission use analysis. In Sadeghi, A.-R., Gligor, V., and Yung, M., editors, *The 2013 ACM SIGSAC conference*, pages 611–622.

Zhauniarovich, Y., Russello, G., Conti, M., Crispo, B., and Fernandes, E. (2014). Moses: Supporting and enforcing security profiles on smartphones. *IEEE Transactions on Dependable and Secure Computing*, 11(3):211–223.

Zheng, M., Sun, M., and Lui, J. C. (2014). Droidtrace: A ptrace based android dynamic analysis system with forward execution capability. In *2014 International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 128–133.