RUHR-UNIVERSITÄT BOCHUM
Horst Görtz Institute for IT Security

Technical Report TR-HGI-2016-003

Evaluating Analysis Tools for Android Apps: Status Quo and Robustness Against Obfuscation

Johannes Hoffmann, Teemu Rytilahti, Davide Maiorca, Marcel Winandy, Giorgio Giacinto, Thorsten Holz

Chair for Systems Security

Ruhr-Universität Bochum
Horst Görtz Institute for IT Security
D-44780 Bochum, Germany

TR-HGI-2016-003
August 2, 2016

RUHR UNIVERSITÄT BOCHUM RUB

hgi Horst Görtz Institut für IT-Sicherheit

Evaluating Analysis Tools for Android Apps: Status Quo and Robustness Against Obfuscation

Johannes Hoffmann[*]
Ruhr-University Bochum

Teemu Rytilahti[*]
Ruhr-University Bochum

Davide Maiorca[†]
University of Cagliari

Marcel Winandy[*]
Ruhr-University Bochum

Giorgio Giacinto[‡]
University of Cagliari

Thorsten Holz[*]
Ruhr-University Bochum

ABSTRACT

The recent past has shown that Android smartphones became the most popular target for malware authors. Contemporary malware families present a variety of features that allow, among others, to steal arbitrary data and to cause significant monetary losses. These circumstances led to the development of many different analysis methods that are aimed to assess the absence of potential harm or malicious behavior in mobile apps. In return, malware authors devised more sophisticated methods to write mobile malware that attempt to thwart such analyses. In this work, we first survey the systems devised to analyze and verify mobile apps and describe the assumptions they rely on to detect malicious content and behavior. We then present a new obfuscation framework that aims to break such assumptions, thus modifying Android apps to avoid them being analyzed by the targeted systems. We use our framework to evaluate the robustness of static and dynamic analysis systems for Android apps against such transformations. In particular, we provide a comprehensive report of the status quo of Android analysis tools against well-obfuscated malware and we demonstrate that most systems could be easily evaded. With our analysis, we point out research problems that should be addressed by future analysis tools and we propose our framework as a possible aid to improve their robustness.

Keywords

Android, DEX bytecode, Obfuscation, Malware, Program Analysis, Survey

1. INTRODUCTION

Malicious software for mobile devices became prevalent in the last few years. While the first of such samples were rather simple and unsophisticated, the complexity of today's malicious apps is steadily increasing. Especially malware for Android-based smartphones has significantly advanced in the recent past. This is a repetition of the evolution of malware we observed for desktop computers: While the very first computer viruses and worms were rather basic, their level of difficulty constantly rose over the years and modern malware like *Stuxnet*, *Flame* or *Regin* impressively demonstrate the sophistication of today's threats. Hence, we also expect that malware for mobile devices will become more mature in the near future.

To counter this development, analysis tools must keep up with the constant evolution of malware. The typical arms race in computer security between attackers and defenders is especially distinctive in this area. Researchers from both academia and industry developed a large number of analysis methods for Android apps in recent years (*e. g.*, [11, 13, 16, 40, 58, 62]). These tools explored many different approaches and they took into account lessons learned from analyzing malware for desktop computers. However, existing analysis methods cannot simply be ported to Android due to many intricacies of the platform (*e. g.*, multiple entry points and a different binary format). Thus, many novel analysis approaches were examined and this led to a huge body of work on this topic.

In this report, we systematically evaluate the robustness of analysis tools for Android apps. Such an empirical evaluation is needed to assess how reliable existing analysis tools are. Moreover, we want to anticipate the next generation of mobile malware that leverages sophisticated obfuscation methods to impede automated analyses. In a first step, we survey existing analysis methods and categorize these approaches according to their main analysis goal. This provides us with a comprehensive overview and leads us to identify three different categories of tools: General-purpose analysis tools, ICC-vulnerability scanners, and behavioral analyzers. We found out that such tools base their analysis on specific basic assumptions. For example, to perform an analysis a tool could rely on (1) having access to the call graph (CG) of the app, (2) being able to follow data flows, or (3) being able to find API function names in the code. To produce an obfuscated app, a viable strategy is thus to systematically thwart these assumptions. For example, we can degenerate the CG such that it does not contain meaningful information, hide all API strings, or conceal object types. The resulting app has the same semantics as the unobfuscated one, but it withstands the analysis tools as their basic assumptions are not fulfilled anymore.

To concretely generate obfuscated apps, we developed a

[*]firstname.lastname@rub.de
[†]davide.maiorca@diee.unica.it
[‡]giacinto@diee.unica.it

framework that specifically targets such assumptions by implementing fine-grained obfuscation strategies, including the following ones: Detection of the analysis environment, entry point pollution, taint analysis bypassing, use-def chains breaking, and type hiding. Contrarily to other free and commercial obfuscators, our framework was not designed to ensure code protection against application tampering or repackaging, nor does it aim to heavily obfuscate commercial applications to make their bytecode less readable. Its main goal is supporting researchers by providing a set of obfuscations techniques with which it is possible to test the robustness of static and dynamic analysis tools.

In addition, our framework allows for developing custom obfuscations with which the user can add and delete instructions, methods and classes. It also gives the possibility to control and obfuscate the application entry-points by writing the necessary changes to the Manifest. Our tool directly operates on DEX bytecode, as this implies that we do not need access to the app's source code to perform our changes.

In summary, we make the following four contributions in this report: (1) We survey existing analysis methods and provide a comprehensive overview of tools. (2) We analyze the basic assumptions underlying existing analysis methods (*e. g.*, ability to reconstruct a CG) and systematically explore how these assumptions can be thwarted (*e. g.*, flattening the CG). Our goal is to produce an obfuscated app with the same semantics as the original one, but which withstands existing analysis tools. (3) We develop a comprehensive framework that employs obfuscation techniques that aim to thwart the assumptions on which the aforementioned tools base their analysis. Our framework is available upon request. (4) We empirically evaluate obfuscated applications by using our framework and find that they severely hamper tools, thus successfully evading both static and dynamic systems. We additionally evaluate how decompilation can be thwarted with obfuscated code. With our analysis, we report on the status quo of current analysis tools for Android apps and propose research guidelines that address the issues we have found during our tests.

2. BACKGROUND & RELATED WORK

We first survey Android analysis frameworks. Then we discuss related work on obfuscation for Android apps and explain why we implemented our own tool. An overview of all mentioned frameworks can be found in Table 1.

2.1 Android Application Analyzers

Prior surveys have given an overview on security research on Android [14] and evaluated the effectiveness of malware detection with dynamic analyses [30]. In contrast, we consider both static and dynamic analysis frameworks and also include general analysis tools. As such, existing approaches of Android application analysis can be categorized into three classes according to their main purpose. In the following, we review the approaches.

2.1.1 Basic and General Analysis Tools

The first class consists of general analysis tools that try to determine specific properties of individual apps. These tools are often used as building blocks for other tools. For example, *Dedexer* [34] and *baksmali* [4] to disassemble DEX files, *ded* [32] and *dex2jar* [2] to convert Dalvik bytecode to Java bytecode, and *Soot* [50] to transform and run code analyses. On the other hand, *WALA* [53] is often used by older tools to analyze transformed Java bytecode instead of directly working on DEX.

AndroGuard [13] is a static analysis framework that includes a decompiler and premade tools for various tasks such as analyzing components and permissions, and determining native code usage. It can also create CFGs and show differences between apps. *SAAF* [22] is a static analysis framework for Android that works on *smali* code, providing program slicing among other common analyses for manual analysis such as generating call-graph and displaying general information about the package.

TaintDroid [15] implements dynamic taint tracking to discover information leakage in apps. This analysis is performed in two basic phases: (1) One or more sources of sensitive data are specified and the corresponding data is marked as tainted; (2) the app is executed and monitored in a modified Dalvik VM that tracks the tainted data flow through the bytecode. When any tainted data reaches a specified sink, *e. g.*, Internet connection, an alarm is raised.

DroidScope [58] is a dynamic binary instrumentation tool that uses a virtualization-based analysis. In contrast to TaintDroid, DroidScope targets both architectural levels of Android: the native Linux and the Dalvik contexts. Besides a modified DVM, it also uses a modified QEMU, which is also instrumented to log specific operations. DroidSope allows extensions through customized analysis plugins, and the authors implemented four different analysis tools: a native instruction tracer, a Dalvik instruction tracer, an API tracer (which logs method invocations of the Android API), and a taint tracker.

More recent research is concentrating on static code and taint analysis, as dynamic analyses may not execute all code paths. *FlowDroid* [18] is a static taint analysis system that provides a precise model of the application lifecycle, and it is context-, flow-, field- and object-sensitive. FlowDroid builds a so-called exploded supergraph based on flow functions that define the data flow on program statements. The decision whether a variable is tainted at a certain point in the code is reduced to a graph reachability problem. *EdgeMiner* [9] aims to provide a comprehensive set of implicit control flows by analyzing the Android framework, which in turn can be used, *e. g.*, by FlowDroid and other tools depending on control flow analysis for more complete coverage. *StaDynA* [61] is a tool that performs static analysis in order to detect the usage of dynamic class loaders and reflections, and to build a method call graph. This graph is then extended by observing the execution of those methods in a modified VM, revealing the real call targets for reflective calls. This allows building callgraphs even when reflections or external class loaders are used to hide functionalities.

Another recent approach is *HARVESTER* [38], whose basic concept is similar to *StaDynA*. The main idea is gathering program slices from a targeted application and creating from those a new APK file for dynamic evaluation. In a first phase, it forms multiple execution paths for interesting slices that are executed on an emulator or on a real device. At the same time, it collects the runtime values of parameters not otherwise available for static analysis, such as encrypted strings and reflective calls. After this analysis phase, it can instrument the original executable to include static values for further processing, making also static analysis feasible even against heavily obfuscated samples.

Table 1: Analysis tools for the Android platform.

		Code Analysis		Requirements			
	Tool	Static	Dynamic	Callgraph	Strings	Based on	Public
General	Soot [50]	✓				—	✓
	AndroGuard [13]	✓		✓	✓	—	✓
	SAAF [22]	✓		✓	✓	—	✓
	TaintDroid [15]		✓			modified DVM	✓
	DroidScope [58]		✓			modified QEMU, modified DVM	✓
	FlowDroid [18]	✓		✓		dexpler, Soot, Heros	✓
	EdgeMiner [9]	✓				—	✓
	StaDynA [61]	✓	✓	✓	✓	modified DVM, AndroGuard	
	HARVESTER [38]	✓	✓			Soot	
ICC vulnerabilities	SCanDroid [19]	✓		✓	✓	decompiler, WALA	
	ComDroid [11]	✓		✓	✓	Dedexer	✓
	CHEX [28]	✓		✓		DexLib, WALA, Stowaway	
	Woodpecker [21]	✓			✓	baksmali	
	DroidChecker [10]	✓		✓	✓	Java Decompiler (JD), ANTLR	
	Epicc [33]	✓		✓	✓	Dare, Soot (Spark, Heros)	✓
	IccTA [27]	✓		✓	✓	Epicc, FlowDroid, ApkCombiner	✓
	Amandroid [54]	✓		✓	✓	modified dexdump	✓
	DIDFAIL [26]	✓		✓	✓	Epicc, FlowDroid	✓
	DroidSafe [20]	✓		✓	✓	Soot	✓
	AppAudit [57]	✓	✓	✓	✓	dex2jar	
Behaviour	ProfileDroid [55]	✓	✓		✓	apktool, adb, strace, tcpdump	
	CopperDroid [40, 48]		✓			QEMU, GDB	✓
	AppProfiler [41]	✓			✓	ded, Fortify SCA, Stowaway	✓
	AppIntent [60]	✓	✓	✓		ded, Soot, JavaPathfinder	
	DroidMiner [59]	✓		✓	✓	AndroGuard	
	AsDroid [24]	✓		✓	✓	dex2jar, WALA	
Sandbox	Andrubis [1]	✓	✓		✓	TaintDroid	✓
	ForeSafe [17]	✓	✓			—	✓
	NVISO [31]	✓	✓			—	✓
	TraceDroid [51]	✓	✓			—	✓
	Mobile-Sandbox [47]	✓	✓		✓	TaintDroid/DroidBox, ltrace	✓

2.1.2 Tools Analyzing ICC Vulnerabilities

Inter-Component Communication (ICC) is another crucial aspect of app security on Android. Apps could be abused via ICC by malicious apps to gain access to sensitive data or to attack their control flow. Messages going to be sent from one app to another could be delivered to an unintended receiver (*e. g.*, via implicit Intents).

SCanDroid [19] is an early static analyzer that checks for data flow security in and between Android apps. This tool concentrates on flows from and to data stores such as content providers, databases, files, and URIs. Furthermore, the tool reasons whether the data flows conform to the permissions the application has and to the given permissions of other applications that are supposed to run on the same device.

ComDroid [11] was the first static analysis tool to examine potential security violations resulting from implicit Intents or publicly made components in Android apps. ComDroid looks for seven different potential ICC vulnerabilities according to the Android component types that are involved with Intent-based ICC. It analyzes disassembled bytecode and identifies the creation of explicit and implicit Intent objects. Furthermore, it follows the control flow to determine Intents that are used in unprotected ICC calls. Besides identifying exported components from the app's manifest, ComDroid also looks in the code for dynamically registered broadcast receivers that listen to implicit Intents or System action strings. *CHEX* [28] focuses on detecting component hijacking vulnerabilities by analyzing disassembled bytecode. *Woodpecker* [21] aims at detecting ICC vulnerabilities in the bytecode of the pre-installed apps on an Android system. *DroidChecker* [10] focuses on finding ICC vulnerabilities that could be exploited by a confused deputy attack by performing an inter-procedural flow and a taint analysis on the decompiled source code to find exploitable data paths.

Epicc [33] is a static analyzer that creates a mapping of ICC app components. It identifies and matches corresponding entry and exit points of ICC according to possible Intent values. The tool is able to analyze the possible Intent values in conditional branches, so it can effectively map call sites to ICC entry points, even across applications. *IccTA* [27] and *DIDFAIL* [26] both combine and improve the analysis tools Epicc and FlowDroid to find privacy leaks across multiple apps that are linked via ICC. *Amandroid* [54] is a generic framework for points-to analysis, and it shares goals with DIDFAIL and IccTA, but provides also other tools besides taint tracking.

DroidSafe [20] improves the analysis of the static source-

to-sink flow by modelling the runtime semantics of critical elements such as strings that are passed as API arguments, native methods, and so forth. Its authors report an increased accuracy compared to *FlowDroid* and *IccTA* combination.

AppAudit [57] detects information leaks by employing dynamic and static analysis. The static part consists of a detailed call graph analysis that looks for suspicious functions by focusing, among others, on reflective API, static fields, and android life cycle methods. The dynamic part executes the suspicious functions bytecode to confirm and extended possible privacy leaks found by the previous analysis. Such execution includes the possibility of approximating unknown bytecode operands to avoid possible problems produced by their presence.

It is reasonable to assume that ICC vulnerabilities will cause a multitude of security violations, as interactions among apps increase. Interestingly, most existing ICC analyzers are based on static analysis, and require the ability of performing string matching to find critical API functions on source code or disassembled bytecode. We can expect that most or all of these tools will fail if apps are obfuscated.

2.1.3 Tools Analyzing Application Behavior

Another group of tools aims at systematically monitoring the behavior of apps. Based on such reports, users can make more informed decisions whether they want to install the app or not.

ProfileDroid [55] generates behavior profiles based on a combination of static and dynamic analyses. It performs a static analysis of the bytecode to search for used Intents, and it dynamically looks at user-generated input events, OS-level system calls, and network traffic usage.

CopperDroid [40,48] performs a dynamic behavioral analysis by logging low-level system calls (*sysctl*) to the Linux kernel and high-level API calls to the Android middleware to infer actions taken by the application without modifying the host system. Based on these call profiles, apps can be compared to those that execute malicious actions. *AppProfiler* [41] follows a similar approach, though resorting to static analysis. It uses a pre-computed mapping of API calls to privacy-relevant behavior, and statically analyzes the decompiled code of apps whether it matches such a profile.

AppIntent [60] analyzes whether external data transmissions in apps can be associated to user intentions. The tool combines static and dynamic methods. First, a static code analysis identifies execution paths leading to external data transmission, and a symbolic execution searches for possible events leading to user actions. Then, a dynamic execution instruments the app to run automatically generated test cases that will show if data transmissions match with user events.

DroidMiner [59] resorts to static analysis to extract Android APIs, and builds a behavioral graph from them. From such graph, it extracts possible malicious behavioral patterns called modalities, and uses them to train statistical classifiers that are used to establish apps' maliciousness.

AsDroid [24] aims to reason about maliciousness based on detecting hidden actions such as network or cellular traffic. It facilitates this by analyzing the descriptions of user-interface elements and trying to connect them to the functionality.

Most of the approaches in this category rely to some extent on static analyses of the code. Hence, we can expect that behavior profiles will not be detectable with these tools when the code of the apps is obfuscated. The only exceptions are *CopperDroid*, *ProfileDroid* and *AppIntent*, which use dynamic analysis to do their analysis. However, the profile of the high-level API calls should be different, *e. g.*, when normal API calls are replaced by reflection calls.

2.2 Android Obfuscators

As we mentioned in the introduction of this paper, the aim of our framework is different to the one of current commercial and free obfuscators. We focus on fine-grained obfuscations that directly target the basic assumptions static and dynamic tools rely on, in order to provide an effective tool to easily test the robustness of analysis tools. Still, for completeness, we provide in this section an overview of the available obfuscators.

DexGuard is probably one of the most powerful commercially available obfuscator so far released for Android. It can rename identifiers and also features advanced transformations, such as utilizing reflections for indirect invokes, or adopting string and class encryption.

Android applications are also often obfuscated with *Proguard*, the open source predecessor of DexGuard. It works on Java code, too, but offers less features and its main purpose is to optimize applications by means of, *e. g.*, string replacements and the removal of unused app code.

Huang *et al.* [23] proposed to apply the algorithms of the Java bytecode obfuscator *SandMark* [49] on Android executables. This was done by transforming DEX to Java bytecode with the *Soot* framework. They tested the transformed applications against repackaging detectors. Unfortunately, as the authors report, this approach has lots of limitations as the conversion from DEX to Java bytecode.

Rastogi *et al.* [39] presented *DroidChameleon*, a framework with which they evaluated what changes to a DEX file produce different results in the detection rates for antivirus programs. They call methods indirectly and encrypt strings next to the encryption of assets. The scope of their work is different to ours, as they implemented simple obfuscation strategies to evade anti-malware systems. Our implemented obfuscations are not only more complex and fine-grained, we also evaluate against more analysis systems.

Protsenko *et al.* presented *Pandora* [36] and evaluated how well antivirus solutions can detect obfuscated Android malware. Their tool applies a multitude of obfuscations such as string encryption, method in-/outlining, and the creation of getters and setters for field access. Their transformations are accomplished with the *Soot* framework, and their main goal is to analyze antivirus robustness against transformation attacks.

Another well-known form of hindering analyses is packing: The original payload is somehow encrypted and only loaded and decrypted when it is being executed [45]. Currently, there are already multiple services (*e. g.*, *Bangcle App Shield*) offering packing for Android applications. *DexProtector* is another commercial tool to pack DEX files, although they also support string encryption. While making static analysis hard, there are ways to dump the original code after decryption from memory (*e. g.*, as described by Shao *et al.* [44]), thus allowing to perform the analysis on the decrypted part. Therefore, such methods should also be combined with other obfuscation strategies for better evasion.

3. PROGRAM ANALYSIS ASSUMPTIONS

We now explain how applications can be altered to thwart analysis attempts. Basing on the survey of existing analysis approaches, we review the basic assumptions these analysis tools rely on to perform their analysis, and discuss obfuscation strategies that will disturb such assumptions.

Our obfuscated apps must fulfill the following three requirements: (1) they must run on Android devices without any required modifications to the OS; (2) they must not be bound to any OS version unless the original app is; (3) they must be obfuscated by directly operating on the DEX bytecode, without transforming them to other forms such as Java bytecode or Java source code.

3.1 Dynamic Analysis Evasion

First, we describe the techniques our framework implements to hinder dynamic analysis systems from producing meaningful results by hampering their analysis attempts as well as hiding data from them.

Analysis Detection: Analysis systems usually resort to a modified instance of the emulator that is shipped with the Android SDK. Malware usually attempts to detect the analysis environment and divert the application's control flow away from its malicious parts. Vidas *et al.* [52] listed a variety of mechanisms to successfully detect such environments. Petsas *et al.* [35] evaluated and tested advanced detection mechanisms that leverage implementation details of QEMU. All these revealing information sources should be changed in custom analysis environments to avoid detection. Still, we found that stock or only slightly modified emulators are often used in analysis systems, and thus such techniques can be easily applied by malware in practice.

Time: As analyses are usually run for a limited time, another common technique to evade dynamic analysis approaches is to perform malicious activities at a certain point in time. Android allows such artificial delays by using for example `Thread.sleep()` or the *AlarmManager*, which we implemented. Analysis frameworks have adapted to this technique and fake the amount of elapsed time, thus circumventing such methods. A more challenging task for analysis systems is detecting expensive computations whose result is used as a requirement for malicious actions. Further, malware could track the computation time for well-known tasks by resorting to external sources such as NTP servers.

Entry Point Pollution: Programs have typically well-defined entry points, *e. g.*, some kind of `main()` method. Because of their event-driven nature, though, Android applications can have a multitude of entry points whose execution is regulated by the Android Framework. Depending on the specifics of these entry points, it is often not clear when and how they are launched, if at all. Thus, a dynamic analysis should invoke all these entry points to generate a good code coverage rate. To complicate analysis attempts, we enable injection of new entry points that are not used by the app, which could either let the application crash, exit, enter an infinite loop, or hamper the analysis in other ways, *e. g.*, setting a flag that gets checked before malicious activities take place.

Anti-tainting: Taint analysis is a powerful analysis technique where data that flows between sources and sinks is tainted with so called *tags* [43]. In a simple example, a framework method which returns the IMEI can be declared as a source and the `send()` method of a socket as a sink. Whenever information is requested from the source, it is marked with a chosen tag and whenever tagged data is sent to a sink, a report is generated. This allows the detection of information leaks through general data modelling. *TaintDroid* [15] was the first tool to offer taint analysis on Android. For our anti-tainting implementation we make use of a list of sensitive sources from previous work by Rasthofer *et al.* [37]. If data from such sources is represented as a numeric value (*e. g.*, a serial number) or a string (*e. g.*, contact information), our injected code automatically "untaints" the data by creating it anew like described by Sarwar *et al.* [42] before it is further processed.

3.2 Static Analysis Evasion

Next, we describe how static analyzers can be evaded. Again, we describe several obfuscation strategies we utilize in order to prevent static analyses. Whereas we only discuss a small excerpt of possible transformations, these techniques are sufficient to hinder the analysis as we later see in Section 4. A comprehensive taxonomy is provided by Collberg *et al.* [12].

Call Graph Degeneration: As Java supports reflections, most direct method invocations can be replaced by indirect ones. A simple example is given in Listing 1. We replace each `invoke-x` and `invoke-x/range` instruction by an indirect call, so that the call graph would be totally degenerated. Only invocations that cannot be replaced this way (*e. g.*, calls to superclass methods due to polymorphism and required invocations that initialize an instance) would be left. Most methods therefore never seem to be called. We have to point out that the code in our example is not well-obfuscated for readability purposes. This will change when we apply additionally techniques described in the following sections.

Breaking Use-Def Chains: In order to track the data flow throughout a program, use-definition (use-def) chains can be used. In DEX code, such chains can easily be built because access to fields and arrays is easily visible by examining the corresponding instructions (*e. g.*, `*put`, `*get`). Therefore, in order to break those chains, we have to make those accesses indirect. Doing so is exemplified in Listing 2. To hide field and array accesses, we replace fields and array calls with their respective reflection methods (lines 8–11 for fields, 14–15 for arrays).

Hiding Types: The previous techniques do not completely hide types, so that they could be easily inferred by an analyst, see Listing 1. Many Reflection APIs accept parameters as `Objects`, letting the virtual machine do the type-checking at runtime. This enables us to get rid of most visible types, except for primitive types and some corner-cases. For example, arithmetic instructions (*e. g.*, `add-int`) and branch-instructions (*e. g.*, `if-eq`) work on non-object registers, which require unboxed, primitive types.

We additionally create new class instances indirectly. All calls to `<init>()` methods are replaced with calls to its counterpart `j.l.r.Constructor.newInstance()`. This still requires a class object, as shown in lines 10 and 13-14 in Listing 1, which is done with the type-revealing `const-class` opcode. These and `const-string` calls are left in the example for readability. Class objects can also be acquired indirectly by using `j.l.Class.forName(String)`, allowing us to remove `const-class` calls. We additionally remove an-

Listing 1: Indirect invokes. Code in line 4 is replaced with semantically equivalent code from lines 6–28.

```
1  sget-object v0, Lj/l/System;.out:Lj/i/PrintStream;
2  const-string v1, "some string"
3
4  invoke-virtual {v0, v1}, Lj/i/PrintStream;.println:(
       Lj/l/String;)V
5
6  const-string v4, "java.io.PrintStream"
7  invoke-static/range {v4}, Lj/l/Class;.forName:(Lj/l/
       String;)Lj/l/Class;
8  move-result-object v4
9  const/16 v8, #int 1
10 const-class v7, Lj/l/Class;
11 invoke-static/range {v7, v8}, Lj/l/reflect/Array;.
       newInstance:(Lj/l/Class;I)Lj/l/Object;
12 move-result-object v6
13 check-cast v6, [Lj/l/Class;
14 const-class v8, Lj/l/String;
15 const/16 v7, #int 0
16 aput-object v8, v6, v7
17 const-string v5, "println"
18 invoke-virtual/range {v4, v5, v6}, Lj/l/Class;.
       getDeclaredMethod:(Lj/l/String;[Lj/l/Class;)Lj/l/
       reflect/Method;
19 move-result-object v9
20 const/16 v12, #int 1
21 const-class v11, Lj/l/Object;
22 invoke-static/range {v11, v12}, Lj/l/reflect/Array;.
       newInstance:(Lj/l/Class;I)Lj/l/Object;
23 move-result-object v11
24 check-cast v11, [Lj/l/Object;
25 const/16 v4, #int 0
26 aput-object v1, v11, v4
27 move-object/16 v10, v0
28 invoke-virtual/range {v9, v10, v11}, Lj/l/reflect/
       Method;.invoke:(Lj/l/Object;[Lj/l/Object;)Lj/l/
       Object;
```

notations that are not required by the virtual machine but that "leak" type information, such as method signatures and debug information.

Applying all these techniques on a method reduces the visible types to only basic Java ones, particularly from the Reflection package. In summary, we access fields and arrays and invoke methods including constructors indirectly over reflection. We also pass parameters as Objects whenever possible, making the type-checking a runtime-only operation. Class objects are also accessed indirectly and we only cast primitive types (and arrays) back to their corresponding types. If required, we also apply Java's auto(un)boxing feature (*e. g.*, converting a primitive int to an Integer). Listing 2 gives an example of how a value is retrieved from a field and stored in a local array without revealing its type.

Bypassing Signature Matching: Some tools identify maliciousness by relying on the occurrence of certain characteristics in an app. We list the most prominent ones and also discuss how we fool such detection mechanisms.

Occurrences: Counting elements such as file sizes, number of classes, fields, methods, or instructions can be used to detect similar programs. However, we can easily avoid this by changing, adding, or removing (unused) parts of a program.

Strings and Literals: In order to save disk space and memory, all unique strings used by an app are stored in an array called *string section* and referenced by an index. This means that all identifiers, types, and strings defined with `const-string` instructions are located in this array. Strings and static numerical values used within a program

Listing 2: Indirect static field and local array access without revealing the object type. Code in lines 4–6 is replaced with semantically equivalent code from lines 8–15.

```
1  const/4 v2, #int 1
2  new-array v1, v2, [Lj/l/String;
3
4  sget-object v0, Lexmpl/Main;.aField:Lj/l/String;
5  const/4 v2, #int 0
6  aput-object v0, v1, v2
7
8  const-class v9, Lexmpl/Main;
9  const-string v10, "aField"
10 invoke-virtual/range {v9, v10}, Lj/l/Class;.
       getDeclaredField:(Lj/l/String;)Lj/l/reflect/Field
       ;
11 move-result-object v8
12 invoke-virtual/range {v8, v9}, Lj/l/reflect/Field;.
       get:(Lj/l/Object;)Lj/l/Object;
13 move-result-object v0
14 const/4 v2, #int 0
15 invoke-static {v1, v2, v0}, Lj/l/reflect/Array;.set:(
       Lj/l/Object;ILj/l/Object;)V
```

do not only give an analyst an overview of the programs intents, but can also be used to detect repackaged applications. Thus, hiding that information is a very effective technique to thwart any analysis attempts that rely on such information. Our tool replaces all instructions that define or reference such information with a method invocation returning the value from an encrypted data structure stored randomly in the app.

Entry Points: Android explicitly declares all possible entry points in the Manifest file of the package. Many tools check the entry points in attempt to detect maliciousness or duplication of known software. Therefore, while renaming classes we also pay attention to rename the entry points when needed. Adding new entry points may also affect the detection. As a common practice to export functionalities to other applications is by using intent filters instead of hard-coded names, we can safely rename all not-exported entry points. Although entrypoints can also be registered during the runtime and thus removed from the manifest file, we did not evaluate that.

4. EVALUATING THE ROBUSTNESS OF ANALYSIS TOOLS

We implemented a framework that is capable of obfuscating arbitrary Android apps with the techniques introduced in the previous section. Our system directly obfuscates DEX code without converting it into an intermediate format (such as JAR). In this section, we test obfuscations produced with our framework on static and dynamic analysis systems.

We have produced self-written samples that exhibit characteristics that should be detected and analyzed by the target systems. Then, we obfuscated such samples by following these guidelines: Most strings, literals and types are hidden; classes, methods, fields and arrays are only accessed indirectly; unnecessary information is stripped from the application (*e. g.*, debug section, specific annotations, unused strings); all types except primitive ones are presented by the most generic one, namely `j.l.Object`, when possible; the manifest file is changed accordingly to the bytecode obfuscations.

The applications are compiled for API level 10 (Android 2.3) and should therefore be supported by all analysis systems. To ascertain that our obfuscated samples were functional, we tested them on a Nexus 5 smartphone running Android 4.4.2 (KitKat) before providing them to the targeted analysis systems. As of September 2015 (during which the framework was being developed) KitKat was the most used version with about 40 % of all devices according to Google [7]. Although Android 5 has now taken the lead, KitKat usage still remains at 32.5 %.

4.1 Implementation

Our framework is written in Java and heavily extends *dexlib*, which is part of the *smali* tool [4]. Our tool accepts a DEX or a complete package (APK) file as input. The obfuscation occurs in three steps: (1) if an APK is given as input, we use *apktool* [56] to extract the package; (2) we directly rewrite DEX bytecode without converting it to intermediate formats. This allows for a more fine-grained control of the VM instructions and registers, and avoids possible crashes or information loss due to the DEX conversions to intermediate formats; (3) the system will create a fully working, obfuscated DEX or APK. Our system is able to fully control all Dalvik instructions, and it is capable of adding, removing or replacing executable elements such as classes, methods, fields, and strings. Refer to the Table 5 in Appendix A for an overview of all our implemented obfuscations. Depending on the application's features, not all techniques are applicable due to some code constraints. We added functionality to dexlib to automatically rewrite parts of a program in order to properly obfuscate it whilst retaining program semantics. Rewriting DEX code is not an easy task, as instructions and registers must carefully be altered as type checking and access flags are enforced everywhere, and some opcodes introduce limitations on how registers can be used.

With Android 4.4 Google introduced a new runtime environment called *Android RunTime* (ART) as a successor to Dalvik, which is now default since Android 5. Therefore, a strongly desired feature is compatibility with both. Even though we are working on bytecode level and the instruction set has not changed, ART still contains a completely new verification code.

Although our goal was to create proof-of-concept samples that could evade analysis tools, we also tested our framework on apps to verify whether they were still working after the obfuscation process. We obfuscated 40 among the most popular apps in Google Play, and manually tested them by interacting with their main functionalities. Of those, 35 apps correctly installed and run. The remaining 5 failed due to some bugs (see Section 7).

4.2 Evaluation of Static Analysis Systems

We begin our evaluation with publicly available static analyzers. All these systems are from academia and are free to download. As our framework flattens the call graph almost completely, we expect that static tools cannot properly analyze the program's control and data flows. They additionally see almost no types, literals, nor strings. All tools relying on such information will likely not be able to produce usable and meaningful results.

Most public static analyzers focus on Inter-Component Communication vulnerabilities. All these tools search for corresponding sinks and sources, *i. e.*, Intents, Receivers and Content Providers. *Epicc* [33] and *ComDroid* [11] are unable to properly analyze data being passed around after obfuscations are applied. The same is true for *FlowDroid* [18]: It is unable to determine sources and sinks because all types are hidden, and aborts the analysis. The same happens with *DIDFAIL* [26], *Amandroid* [54], and presumably with other static flow analyzers. We did not test *DroidSafe* due to its really high system requirements (recommended to have at least 64 GB of memory). Because of the implicit control flow instructions that are used in our obfuscations, we stop information leaks that might be detected by the precise control flow handling of *EdgeMiner* [9]. All tested tools generate no results on our obfuscated test samples. The only information available to these tools is the information defined in the manifest file.

All the other public static analyzers failed at gathering information from our obfuscated test apps. For example, *SAAF* [22] is not able to retrieve meaningful information from generated program slices. Analysis results also miss relevant information and cannot be used to understand the program's semantics. Our implemented obfuscations also break tools that rely on Java decompilation, such as *DroidChecker* [10]. We provide more details about this topic in Section 5. *StaDynA* [61] is able to construct call-graphs for obfuscated applications as expected, but due to a bug in AndroGuard [13] it fails to do it for some samples. In general, their approach might be used to form proper call-graphs for further analysis, but we cannot provide detailed results.

Our results show that automatically applied obfuscation to programs completely defeats static analyzers. Most information accessible by them is only of generic value, and does not lead to informative analysis reports. The heavy use of reflections can be flagged suspicious by such tools, but it cannot be used solely for tagging apps as malicious due to its wide-spread use. Most of the benign applications resort to reflective obfuscations to avoid being easily analyzed.

4.3 Evaluation of Dynamic Analysis Systems

In these experiments, we test the capabilities of dynamic systems to detect evasive behaviors under obfuscation. Because of their basic properties, such behaviors should easily be detectable.

To test such systems, we wrote four applications that exhibit malicious and evasive behavior which can be easily developed to thwart dynamic analysis. Then, we obfuscated them and made them analyze by five well-known dynamic analysis services by the time of writing. If a tested dynamic system failed at detecting such attacks, it would most likely fail with more complex evasion strategies too. All our applications display a "Hello World" activity while invoking additional functions, which we describe now:

- *Direct*: this app creates three threads and each of them performs suspicious actions: The first one sends an SMS; the second one sends the Browser's search terms over a TCP socket to one of our servers; the third one acts like the second one, but steals the IMEI.
- *Sleep*: This app calls `Thread.sleep()` for 5 mins before sending an SMS.
- *Alarm*: This works similarly to *Sleep*, but uses *AlarmManager* for delaying the sending.
- *EmuDetect (ED)*: this app attempts to detect whether it is running in an emulator before sending an SMS and the IMEI.

Table 2: Results for dynamic analysis services against analysis evasion techniques.

Vendor	Direct	Sleep	Alarm	ED	Taints
Andrubis	✓	↯	✓	↯	↯
ForeSafe	↯	↯	↯	↯	*n. a.*
Mobile Sandbox	✓	✓	✓	↯	↯
NVISO	✓	↯	↯	↯	↯
Tracedroid	✓	↯	✓	↯	*n. a.*

The aforementioned actions are often used by malware and should therefore be reported by analysis systems. Our emulator detection also is rather simple and it is well-detectable. We check against well-known IDs that default to known values in the emulator: *Build.Hardware*, *Build.Model*, *Build.Product*, the network operator's name, *IMEI*, and the *Android ID*. Additionally, all applications make use of anti-tainting techniques.

All tests are implemented in a straightforward fashion by using standard APIs and should thus be easily detectable. For example, this means that a system should be able to detect if the *SmsManager* was accessed and an SMS was directly sent with values directly declared in that method.

We also include the *Google Play Services* library to our *EmuDetect* test to check whether the Play services are correctly set up on the device and whether a connection to them can be established. If that succeeds, we retrieve the *Android Ad ID*. The last check is not complex, but requires a fully set up Google Play environment. Even real hardware devices fail this test if the service is not properly updated and set up with a valid Google account.

We show a summary of the results of our tests in Table 2. Satisfying analysis results—meaning the analysis system was resistant to our modifications—are marked with a "✓". If the system fails at retrieving information, we mark it as "*n. a.*". If provided results for that application do not contain hints for suspicious behavior (such as simply marking it as "unsuspicious"), we mark them in the table with a "↯". All mentioned frameworks are also listed in Table 1 in Appendix A.

The analyzed services base their taint tracking on *TaintDroid* [15], which should detect possible leaks and report them. If leaked information is being sent back to us and the service report does not provide information about it, but does so when the application is not obfuscated, we know that our implemented obfuscation techniques are successfully evading taint analyses. Lost taints are marked with a "↯" in the "Taints" section of the table. If tainted data is not specifically marked in reports, but is for example contained in network dumps, we mark it as "*n. a.*", as no tags have been added in the first place. The results obtained by the tested analysis systems presented in Table 2 demonstrate many shortcomings of existing analysis methods. We now provide more details about them.

Andrubis [1] displays results on the service webpage, by providing all the network activity. The service assigns a malicious value ranging from 0 (likely benign) to 10 (likely malicious) to tested applications. For our samples, such scores were always towards malicious. Andrubis successfully analyzed the *Direct* and *Alarm* samples, but failed for the *Sleep* and *EmuDetect* samples. Taint tags are not retained, even though the report contains a section labeled "Data leaks". We found that apart from the IMEI, no other identifiers were changed and no valid Google Account is set up.

Mobile-Sandbox [47] combines static and dynamic analysis to identify malicious functionality in apps. A static analysis checks for malware, determines required permissions, and identifies possible entrypoints. Dynamic results provided by it look promising. It is the only analyzer that is able to correctly analyze both delaying samples, but also fails the emulator detection (only because no valid Google account is available). It additionally also fails the anti-taint test. As Mobile Sandbox is based on *DroidBox* [3], we did not evaluate it separately.

NVISO [31] was also able to analyze all four samples and provided nice results (including a screenshot) for the *Direct* sample, by ranking it as "confirmed malicious". All the information is available, although some can only be found in the provided PCAP file. The report does not directly show that browser searches or the IMEI have been leaked. This is caused by our implemented obfuscations, as other reports contain such information. The connection to our server is also listed. All the other applications are ranked with "no malicious activity detected" and the sending of the SMS goes undetected. The used emulator only reports a changed IMEI.

ForeSafe [17] analyzed our samples and provided us with a screenshot of our running application, meaning that the app could be successfully started. The report of the dynamic analysis rates our app with a "No Risk Detected". There was no mention about any of our performed activities. We even got three connections to our server, so the app seems to be started multiple times. Additionally, the IMEI and other identifiers were unchanged. A quick static check performed before the dynamic part also states that no suspicious elements could be detected. We observe that ForeSafe is the only system that failed at detecting many of the activities even in non-obfuscated samples, except for the SMS sending. Since nothing risky was detected, we checked ForeSafe against our non-obfuscated sample and then at least the sent SMS was detected with the correct destination and text part (but also did not flag the app as suspicious). No identifiers are changed during our tests, making the emulator easily detectable.

Tracedroid [51] reports for our test samples contain a lot of information and provide a complete trace of the programs. Even calls done reflectively are listed with the used parameters, making it possible to see all the invoked methods, accessed classes, and fields. Reports are provided in text files for each thread containing the execution trace, and they completely reveal what happened while executing the application. However, the systems fails at detecting activities performed by our *Sleep* and *EmuDetect* applications. A screenshot next to a (flattened) call graph (PDF) and a network dump are also provided. No strings that identify the emulator are changed, so the service is easily detectable and malicious activities can be suppressed. Declared services in the Manifest file are also started, even if they are not accessed from the application itself.

In addition to the analysis systems listed in Table 2, we tried to test other known systems, but failed to do so for various reasons. For example, the currently available version of *CopperDroid* [40, 48] was not able to analyze our applications as it required apps compiled against a rather old Android version 2.2. According to its developers, *Copper-*

Table 3: Decompilation results for different tests.

	GO1	GO2	CHO	CT	DC	BC	SM	RSA	MC
jadx	ϟ	ϟ	ϟ	ϟ	ϟ	ϟ	✓	(✓)	(✓)
dex2jar	ϟ	W	ϟ	✓	✓	ϟ	M	(✓)	I
Dare (JD-GUI)	ϟ	W	ϟ	I	I	ϟ	✓	(✓)	I
Dare (Soot)	M	✓	ϟ	I	✓	ϟ	✓	(✓)	(✓)

Droid supporting newer Android versions is currently under construction [48].

5. EVALUATION OF DECOMPILERS

In this section, we provide an insight into using our framework to apply obfuscations that target decompilers. In particular, we use fine-grained techniques that exploit differences between Java source code and Dalvik bytecode. Such strategies are developed to make decompilers crash, thus not revealing any useful information about the code.

5.1 Decompilation

In this section, we report the results of the test on four freely available decompilers for Android applications: *jadx*, *dex2jar* with *JD-GUI*, *Dare* with *JD-GUI* and *Soot*. These decompilers try to transform DEX code to a Java-equivalent, and are also adopted by some analysis tools. We use our framework to inject code constructs that are allowed in DEX, but that are not permitted in Java. This is done to see how a decompiler would react when trying to parse such constructs.

We developed nine test cases, each of them implementing a different construct. Condensed results can be found in Table 3, where we use the following notations: If the test case is handled correctly, we assign a "✓" ; we assign a "(✓)" if the code is almost correct, and can very easily be fixed by an analyst; invalid code will be marked with an "I", meaning that such code cannot be recompiled without further modifications; syntactically correct code that reflects different semantics will be marked as "W"; if instructions are missing in the Java code, we note that with an "M" (meaning that code can be hidden from such tools); if the decompiler is not able to obtain any results for a method, but raises an exception (when that happens, decompilers often print a dump of the DEX code), we mark it with "ϟ".

Goto Instruction (GO1/GO2): The `goto` instructions can pose problems to decompilers, as there is no direct equivalent in Java, and the code can jump arbitrarily inside a method (including between catch handlers). Imagine a method as shown in Listing 3, where the invoked method will normally never throw an exception (with rare exceptions). Still, the invocation is covered by a try-item which specifies two handlers. The problem here is that both handlers refer each other and form a loop (line 3–4). While executing the shown method, these handlers are never touched. In Java, each handler forms a new scope and cannot interfere with other handlers, making this construct impossible. All decompile crash with this test (GO1) with the exception of *Dare (Soot)*, which completely ignores the try-catch clause. In the second test (GO2), we tested how decompilers handle the `goto/32` instruction referring to itself. Example code is shown in Listing 4. The code jumps to the invocation of a method and then enters an infinite loop (line 2), which is completely legit in DEX. Only *Dare (Soot)* works correctly by substituting the instruction with an empty *while (true)* loop. *Dare (JD-GUI)* and *dex2jar* call the method inside a *while (true)* loop and *jadx* crashes with an exception.

Listing 3: Misuse of *goto* instruction, example (1).

```
1 0x00: invoke-static/range {}, Lsome/Class;.someMethod
      :()V
2 0x03: goto 0006 // +0003
3 0x04: goto 0005 // +0001
4 0x05: goto 0004 // -0001
5 0x06: return-void
6  catches       : 1
7   0x00 - 0x03
8   Ljva/lang/NullPointerException; -> 0x05
9   Ljva/lang/RuntimeException; -> 0x04
```

Listing 4: Misuse of *goto* instruction, example (2).

```
1 0x00: goto 0004 // +0004
2 0x01: goto/32 #00000000
3 0x04: invoke-static/range {}, Lsome/Class;.someMethod
      :()V
4 0x07: goto 0001 // -0006
5 0x08: return-void
```

Catch Handler Order (CHO): Imagine a try-catch clause that catches two exceptions in the same hierarchy, and the second handler catches a subtype of the first handled exception. If we reverse the handler order, the second handler will never be executed, as the first one always handles the thrown type. A matching handler is searched based on the type in the order of declaration. The Java compiler enforces correct ordering, as otherwise dead code would be generated. A wrong order will therefore result in invalid Java code if decompiled. For our example in Listing 5, the correct result would be to simply remove the second handler. As it turns out, no tool is able to decompile the method and they all abort with an exception.

Listing 5: Wrong exception handler order.

```
1 0x00: const v0, #int 42
2 0x03: invoke-static/range {}, Lsome/Class;.someMethod
      :()V
3 0x06: return-void
4 0x07: add-int/lit16 v0, v0, #int 1
5 0x09: goto 0006 // -0003
6 0x0a: add-int/lit16 v0, v0, #int 2
7 0x0c: goto 0006 // -0006
8  catches       : 1
9   0x03 - 0x06
10  Ljava/lang/Exception; -> 0x0a
11  Ljava/lang/RuntimeException; -> 0x07
```

Clashing Types (CT): In DEX, some registers can have different types depending on the usage. The instruction `const v0, 0`, for example, assigns the DEX type *ZERO* to register *v0*. This type can then be interpreted in multiple ways: for reference types it refers to a *null* reference, for booleans to false, and for primitive literals to 0. In our test, we put such literal to fields with different types. As a result, only *dex2jar* correctly handles the types, while *jadx* quits with a thrown exception. *Dare (JD-GUI)* tries to assign 0 to a boolean field, while *Dare (Soot)* casts 0 to an Object.

Dead Code (DC): Dead code refers to code that cannot be reached at all. For our test, we inject valid code after `return` instructions. Ideally, such code does not show up

in decompiled code. Results show that *jadx* quits with an exception. *dex2jar* and *Dare (Soot)* correctly handle the code. *Dare (JD-GUI)* returns completely invalid code with wrong assignments such as *this = 5;*.

Bogus Code (BC): For this test, we injected an unused class with a method containing invalid instructions. Executing such a method would cause a crash. As the class is unused, the application will still be installable and executable. A subtle example would be, *e. g.*, a method invocation using a register holding an uninitialized reference. We quickly found code where all decompilers bail out with an exception. With the sole exception of *jadx*, all tools stopped working completely while analyzing the method in question, skipping the rest of the program and providing no output in the end. *jadx* only threw an exception for the methods in question, but proceeded with other code.

Synthetic Modifier (SM): We also evaluated what happens if we add the *synthethic* modifier to classes, methods, and fields. Technically, the modifier hints that the construct was generated by the compiler and has no counterpart in the source code. Bridge methods are treated as such. All decompilers correctly handle the code (thus ignoring the synthetic modifier—*jadx* also points out its presence by adding a comment to the class)—with the exception of *dex2jar*, which completely hides the content of the class or method.

Removing Signature Annotations (RSA): DEX stores information about types in generic containers inside *Signature* annotations, when needed. This is done in order to deliver the information about what is being contained, *e. g.*, in a `List<T>` for debuggers. When removing signatures, the tested decompilers are unable to infer types correctly. This causes missing type information with all tested compilers, but may also break applications leveraging that information during runtime.

Modifier Changes (MC): Modifying access flags can also cause different results with different decompilers. In Java, an enumerated type extends `j.l.Enum<T>` implicitly, and thus cannot extend any other class. However, in DEX it is possible to declare any class with an *ENUM* access flag without severe runtime side effects. When decompiling such bytecode, *dex2jar* will mark the class as an enum and omit the superclass information completely. *Dare (JD-GUI)* has the same behavior. *jadx* will still show super classes correctly, like *Dare (Soot)* also does.

Judging from the results presented above, it is clear that all decompilers fail for at least four out of nine test cases and no test case is handled by all tools with the exception of simply removing type information. Automatically creating such code reliably hinders the analysis of an obfuscated application if the analysis depends on decompiled code. The most robust decompiler based our tests is *Dare (Soot)*.

6. PERFORMANCE EVALUATION

We now provide insight into the performance of apps modified by our framework. Although not the main focus of this paper, we still address this topic because obfuscated malicious or "protected" apps should not introduce huge slowdowns, as users would reject such apps. We therefore evaluate how our modifications affect the execution speed and discuss whether such obfuscation techniques are acceptable from a user's point of view.

We start with artificial benchmark results to get an idea of how large the slowdown can be. We wrote four small test cases for testing this: (1) Open 200 sockets and immediately close them again; (2) create 200 files; (3) create 100 Java processes that execute the "id" command in a shell while reading its output; (4) loop over an array of 10,000 primitive integers, and sums them up. Since operations occur on primitive types, they are not obfuscated. The loop condition check is done indirectly, though. Each loop iteration therefore performs a corresponding method call instead of just executing the original `array-length` instruction.

Each test is performed exactly twelve times. The best and worst timings are discarded, and an average is taken from the remaining ten values. The results for two different smartphones are listed in Table 4. All of our described obfuscation techniques in Section 3 have been applied. The Galaxy Nexus runs Android 4.3 with Dalvik and the Nexus 5 runs Android 5 with enabled ART runtime.

Table 4: Benchmark results. Values in seconds.

Device	Obf.	Socket	File	Process	Array
Nexus 5/ART		0.7756	0.1581	2.4756	0.0127
Nexus 5/ART	✓	1.4549	0.9422	2.6515	0.4890
Galaxy Nexus		1.5791	0.1972	1.6423	0.0375
Galaxy Nexus	✓	1.9243	0.8896	2.7481	0.8598

What can be seen is that the overhead can vary by a huge margin, depending on the test. The least overhead is given on the Nexus 5 for the process creation test. The test is barely slower under obfuscation than the original one. The second worst overhead is introduced for the file creation test on the same device. This test is almost six times slower in the obfuscated version. The worst slowdown is introduced for the array-test. This test clearly shows how huge a slowdown can be if only one single instruction is exchanged with a semantically equivalent call over reflection. On the Nexus 5 the obfuscated sample is about 39 times slower. Modifying instructions in loops that would be efficiently optimized by the runtime, can lead to enormous slowdowns.

As these tests are purely artificial, we also evaluated what a user experiences while using an obfuscated app. For this test case, we took two identical Nexus 5 phones, and installed the original app in one and the obfuscated in another. Most apps do not perform heavy operations on the main thread, which is also responsible for the GUI rendering, making the perceived slowdown unnoticeable if we simultaneously perform the same actions on those two devices. What causes huge slowdowns are recursive operations, *e. g.*, parsers that parse JSON objects retrieved from the Internet. Apps also feel slower if many instructions are triggered on input events. Firefox, *e. g.*, automatically starts suggesting URLs. Such operation is very expensive, even when unobfuscated. These actions are noticeable by the user, but can easily be blacklisted in order to avoid the slowdown—at the cost of unobfuscated program code.

Besides runtime overhead, we also evaluated how our implemented obfuscations affect the size of obfuscated applications. As several instructions are replaced with multiple ones, the size can quickly grow. We found that the application's code increases by approximately 20 % on average.

We deem both the runtime and size overhead acceptable, as the user's experience is only briefly impaired. Still, performance penalties caused by, *e. g.*, method and string lookups,

(especially in loops) do introduce slowdowns which can further be optimized by means of (more) caching.

7. DISCUSSION AND LIMITATIONS

In the following, we discuss the limitations of our tool and how they can lead to semantically different execution or even broken code. We also explain how limitations could be fixed in future versions of our approach. Renaming classes, methods, and fields is tricky for event-driven applications, as they may be executed by external parts out of our control. For Android, the framework itself executes well-defined entry points which can be obfuscated if the definition is appropriately changed in the Manifest. For example, an application may leverage hard-coded entry points. One way to tackle the problem is through blacklisting.

The same is true for code with *implicit dependencies*, which expect, *e. g.*, fields or methods to have certain names. While we can control the code of our application, we may still be restricted by the *implicit* API choices of the framework or of other libraries, *e. g.*, `Bundle.CREATOR` field. Without analyzing the behavior of the app, it is hard to know which elements can be freely changed.

During our research, we encountered multiple applications calling methods from native libraries, causing the aforementioned problems. In order to make those invocations work correctly, we would need to instrument all these calls through JNI interfaces to point them to renamed methods.

Some bugs occurred within obfuscated apps, including rejections by the Android verifier and crashes during install. However, that is expected for some corner cases when working with a prototype—we are progressively fixing our framework to solve such bugs. As our tool will be released, contributors can also modify some of its parts to improve its functionalities. Additionally, one can exclude problematic application parts by means of a blacklist in order to skip them while obfuscating.

Android also has a quite limited amount of available stack memory by default. This can be changed for new threads, but not for the framework threads that execute most the of application's code. If all invocations are replaced with indirect ones, for each called method two additional methods are put onto the call stack. This can cause the stack to overflow for some deep call paths. Notable examples are libraries that do recursive parsing for, *e. g.*, JSON objects.

By utilizing static or dynamic instrumentation frameworks and dynamic analyzers in general, it is always possible to deobfuscate a program as the runtime semantics must stay the same. Applied obfuscations in general make this task more time consuming.

7.1 Skipped Obfuscation Steps

Although our tool is able to add bogus entry points into an application in question, we did not evaluate how tools deal with it. Static analyzers are already unable to obtain any meaningful results about the program semantics with just our other techniques enabled. While they still can analyze generic aspects of applications, the need to further distract them with additional entry points is unnecessary. Dynamic analyzers also did not require that feature, as all but one emulator can easily be detected with simple tests.

We did not address the injection of *opaque predicates* [12], which can be used to amplify the path explosion problem in *multipath execution* [29] and dynamic *symbolic execution* [46]. This is because the two existing prototypes for symbolic execution, *SymDroid* [25] and *ACTEve* [5, 6], are still very rough prototypes that would require significant effort to make them usable for our purposes. To the best of our knowledge, *HARVESTER* is the only framework that implements multipath execution, but it is currently not available.

We also omitted techniques like method merging and inlining, as well as moving methods and fields around. Dynamic analyzers are not affected by this effect, and static ones are already blind with respect to the performed obfuscations. We also do not obfuscate literal values of `0` and `1`, due to their varying semantics.

8. CONCLUSION

In this paper, we evaluated how current analysis tools can cope with heavily obfuscated apps. To do so, after having retrieved all publicly available static and dynamic analyzers, we have developed a framework for automated obfuscation of Android apps. Our framework implements fine-grained obfuscation strategies that can be used as test benches for evaluating the robustness of analysis tools.

In our analysis, we targeted both static and dynamic analyzers, including also decompilers. Our test results let us conclude that many analysis tools are not capable of analyzing obfuscated apps in a satisfying manner.

The worrying aspect is that the code modifications as described in Section 3 can be automatically applied on arbitrary apps without needing access to source code. Malicious software can easily piggyback obfuscated apps, and clandestinely execute their payload while most detection systems remain blind. To this end, the recent work by Zhauniarovich *et al.* [61] and Rasthofer *et al.* [38] shows a great promise, helping to bring static analysis methods again usable even against heavily obfuscated apps.

Availability: To foster research on analysis of obfuscated apps and increase the robustness of existing methods, we are offering our framework for research purposes upon request.

9. REFERENCES

[1] Anubis – Malware Analysis for Unknown Binaries. `https://anubis.iseclab.org/`.

[2] dex2jar. `http://code.google.com/p/dex2jar/`.

[3] DroidBox. `http://code.google.com/p/droidbox/`.

[4] smali. `http://code.google.com/p/smali/`.

[5] S. Anand and M. J. Harrold. Heap cloning: Enabling dynamic symbolic execution of java programs. In *ASE*, 2011.

[6] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *SIGSOFT FSE*, 2012.

[7] Android Developers. Platform Versions, June 2014. `http://developer.android.com/resources/dashboard/platform-versions.html`.

[8] Anonymous. To be released. Technical report, 2015.

[9] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[10] P. P. Chan, L. C. Hui, and S. M. Yiu. DroidChecker: Analyzing Android applications for capability leak. In

ACM Conference on Security and Privacy in Wireless and Mobile Networks, WISEC '12, 2012.

[11] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *Intern. Conf. on Mobile Systems, Applications, and Services (MobiSys)*. ACM, 2011.

[12] C. Collberg, C. Thomborson, and D. Low. A Taxonomy of Obfuscating Transformations. Technical report, University of Auckland, 1997.

[13] A. Desnos and G. Gueguen. Android: From reversing to decompilation. *In Proc. of Black Hat Abu Dhabi*, 2011.

[14] W. Enck. Defending users against smartphone apps: Techniques and future directions. In *7th International Conference on Information Systems Security (ICISS)*. Springer, 2011.

[15] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI*. USENIX Association, 2010.

[16] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions Demystified. In *CCS*. ACM, 2011.

[17] ForeSafe. ForeSafe Online Scanner. `http://www.foresafe.com/scan`.

[18] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel. Highly precise taint analysis for Android applications. Technical Report TUD-CS-2013-0113, TU Darmstadt, 2013.

[19] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. SCanDroid: Automated security certification of Android applications. Technical Report CS-TR-4991, Department of Computer Science, University of Maryland, College Park, Nov. 2009.

[20] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard. Information Flow Analysis of Android Applications in DroidSafe. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[21] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock Android smartphones. In *NDSS*, 2012.

[22] J. Hoffmann, M. Ussath, T. Holz, and M. Spreitzenbarth. Slicing Droids: Program Slicing for Smali Code. In *SAC*. ACM, 2013.

[23] H. Huang, S. Zhu, P. Liu, and D. Wu. A framework for evaluating mobile app repackaging detection algorithms. In *TRUST*. Springer, 2013.

[24] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang. Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014.

[25] J. Jeon, K. K. Micinski, and J. S. Foster. SymDroid: Symbolic Execution for Dalvik Bytecode. Technical report, Department of Computer Science, University of Maryland, College Park, 2012.

[26] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer. 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*, SOAP '14. ACM, 2014.

[27] L. Li, A. Bartel, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. I know what leaked in your pocket: uncovering privacy leaks on android apps with static taint analysis. *CoRR*, abs/1404.7431, 2014.

[28] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: Statically vetting Android apps for component hijacking vulnerabilities. In *CCS*. ACM, 2012.

[29] A. Moser, C. Krügel, and E. Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2007.

[30] S. Neuner, V. V. der Veen, M. Lindorfer, M. Huber, G. Merzdovnik, M. Mulazzani, and E. R. Weippl. Enter sandbox: Android sandbox comparison. In *Proceedings of the IEEE Mobile Security Technologies Workshop (MoST)*. IEEE, 2014.

[31] NVISO. NVISO ApkScan – Scan Android applications for malware. `http://apkscan.nviso.be/`.

[32] D. Octeau, W. Enck, and P. McDaniel. The ded Decompiler. Technical Report NAS-TR-0140-2010, Networking and Security Research Center, Pennsylvania State University, Sept. 2010.

[33] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective Inter-component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis. In *USENIX Security Symposium*, 2013.

[34] G. Paller. Dedexer. `http://dedexer.sourceforge.net/`.

[35] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis. Rage against the virtual machine: hindering dynamic analysis of Android malware. In *EUROSEC*. ACM, 2014.

[36] M. Protsenko and T. Müller. Pandora applies non-deterministic obfuscation randomly to android. In *MALWARE*. IEEE, 2013.

[37] S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning approach for classifying and categorizing Android sources and sinks. In *2014 Network and Distributed System Security Symposium (NDSS)*, 2014.

[38] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden. Harvesting runtime data in android applications for identifying malware and enhancing code analysis. Technical Report TUD-CS-2015-0031, TU Darmstadt, 2015.

[39] V. Rastogi, Y. Chen, and X. Jiang. Catch Me If You Can: Evaluating Android Anti-Malware Against Transformation Attacks. *IEEE Transactions on Information Forensics and Security*, 2014.

[40] A. Reina, A. Fattori, and L. Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct Android malware behaviors. In *Proceedings of the 6th European Workshop on System Security (EUROSEC)*, 2013.

[41] S. Rosen, Z. Qian, and Z. M. Mao. AppProfiler: A flexible method of exposing privacy-related behavior in Android applications to end users. In *Proc. of the*

3rd ACM Conf. on Data and Application Security and Privacy, CODASPY. ACM, 2013.

[42] G. Sarwar, O. Mehani, R. Boreli, and M. A. Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on Android-based devices. In *SECRYPT*, 2013.

[43] E. J. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2010.

[44] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang. Towards a scalable resource-driven approach for detecting repackaged android applications. In *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 2014.

[45] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee. Impeding Malware Analysis Using Conditional Code Obfuscation. In *NDSS*, 2008.

[46] D. X. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *4th International Conference on Information Systems Security (ICISS)*. Springer, 2008.

[47] M. Spreitzenbarth, F. C. Freiling, F. Echtler, T. Schreck, and J. Hoffmann. Mobile-Sandbox: Having a Deeper Look into Android Applications. In *SAC*. ACM, 2013.

[48] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[49] Univ. of Arizona - Dept. of Computer Science. SandMark: A Tool for the Study of Software Protection Algorithms. `http://sandmark.cs.arizona.edu/index.html`.

[50] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java Bytecode Optimization Framework. In *Proc. of the Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON. IBM Press, 1999.

[51] V. van der Veen and C. Rossow. Tracedroid. `http://tracedroid.few.vu.nl`.

[52] T. Vidas and N. Christin. Evading Android runtime analysis via sandbox detection. In S. Moriai, T. Jaeger, and K. Sakurai, editors, *ASIACCS*. ACM, 2014.

[53] WALA developers. T.J. Watson Libraries for Analysis (WALA).

[54] F. Wei, S. Roy, X. Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014.

[55] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. ProfileDroid: Multi-layer profiling of Android applications. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, Mobicom '12. ACM, 2012.

[56] R. Wisniewski. android-apktool – A tool for reverse engineering Android apk files. `http://code.google.com/p/android-apktool/`.

[57] M. Xia, L. Gonga, Y. Lyu, Z. Qi, and X. Liu. Effective real-time android application auditing. In *36th IEEE Symposium on Security and Privacy, San Jose, CA*, 2015.

[58] L. K. Yan and H. Yin. DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In *Proceedings of the 21st USENIX Security Symposium*. USENIX Association, 2012.

[59] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. In *Proceedings of the 19th European Symposium on Research in Computer Security (ESORICS'14)*, September 2014.

[60] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. AppIntent: Analyzing sensitive data transmission in Android for privacy leakage detection. In *CCS*. ACM, 2013.

[61] Y. Zhauniarovich, M. Ahmad, O. Gadyatskaya, B. Crispo, and F. Massacci. StaDynA: Addressing the Problem of Dynamic Code Updates in the Security Analysis of Android Applications. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, CODASPY '15, 2015.

[62] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *NDSS*, 2012.

APPENDIX

A. IMPLEMENTED OBFUSCATIONS

Table 5 gives an overview of the features of our framework. The entries marked with "†" were implemented, but not analyzed as a part of this paper. *Package flattening* removes the class hierarchies from obfuscated binaries. *Method removal* and *method call removal* remove unnecessary methods and their invocations, such as *toString()* methods and calls to logging facilities, to avoid leaking information to analysts. *Access flags changes* can be used to avoid decompilation with tools that fail for such binaries.

Table 5: Implemented obfuscation techniques († = not evaluated)

	Technique	Useful against: Static	Useful against: Dynamic
Data	Constant Hiding	✓	
	Package Flattening †	✓	
Layout	Identifier Renaming	✓	
	Annotation Removal †	✓	✓
	Debug Information Removal †	✓	✓
	Removal of Unused Strings †	✓	
	Method Call Removal †		✓
	Method Removal †		✓
Control	Indirect Invocations and Accesses	✓	✓
	Entry Point Pollution †	✓	✓
Preventive	Dynamic Analysis Evasion		✓
	Anti-tainting	✓	✓
	Access Flag Changes †	✓	