COLLEGE MATHS, TELECOMS
DOCTORAL INFORMATIQUE, SIGNAL
BRETAGNE SYSTEMES, ELECTRONIQUE

CentraleSupélec

Thèse de doctorat de

CentraleSupélec

École Doctorale N° 601
Mathématiques, Télécommunications, Informatique, Signal, Systèmes, Électronique
Spécialité : *Informatique*

Par

Jean-Marie MINEAU

The Woes of Android Reverse Engineering: from Large Scale Analysis to Dynamic Deobfuscation

Les difficultés de la rétro-ingénierie Android: de l'analyse large échelle au dé-brouillage dynamique

Thèse présentée et soutenue à Rennes, le 2025-12-09
Unité de recherche : IRISA
Thèse N°: 2025CSUP0006

Composition du jury :

Président :	Olivier Barais	Professeur des Universités	Université de Rennes
Rapporteurs :	Vincent Nicomette	Professeur des Universités	INSA de Toulouse
	Julien Signoles	Directeur de Recherche	CEA LIST
Examinateurs :	Simone Aonzo	Maître de Conférences	Eurecom
Dir. de thèse :	Jean-François Lalande	Professeur des Universités	CentraleSupélec
	Valérie Viet Triem Tong	Professeure	CentraleSupélec

n

Acknowledgements

First of all, I would like to express my gratitude to Vincent Nicomette and Julien Signoles, who agreed to be the *rapporteurs* for this thesis, as well as to Olivier Barais and Simone Aonzo for being part of the Jury.

I'm also grateful for the support of my thesis advisors, Jean-François Lalande and Valérie Viet Triem Tong. They trusted me to explore my strange ideas, reminded me to take things one at a time when I was overwhelmed by the number of issues one encounters when working with Android, and overall gave me great advice during those three years. Now that I have finished my PhD, I can confirm that the curve of the evolution of the morale of the PhD student Jean-François drew me is pretty accurate.

I want to thank Marie Babel and Jacque Klein for being part of my *Commité de Suivi Individuel*, and my colleagues of the 5th floor, from both the PIRAT\‘); and the SUSHI team. Those 3 years were fun, the croissants were good, and the conversations were weird. What more could you ask for? And I wish courage and good luck to those on the tail-end of their PhD, you're almost there!

As promised, I thank Amelie for proofreading my French summary. I forgive you for the typos you missed. Not as promised, I also thank you for the moral support and all the cute animal pictures exchanged during the last 3 years.

I also want to thank the *Pains-Perdus*, some of you may or may not have pushed me into doing a PhD, and you managed to endure my many rants about Android.

Last but not least, I thank my family. Antoine for the good food every time I'm in the area. Kpu for all the sword fighting when you were in Rennes, now that you have the HEMA virus, my job is done. My mother, for giving me my first taste of infosec, even if it was definitely not on purpose. Bypassing parental control on a wifi router is a great way to start. And my father, for showing a 12-year-old me how to read and modify the source code of Battle for Wesnoth, who knew this first step in the world of reverse engineering would lead to a PhD?

Thank you.

Table of Contents

Index of Figures

Index of Tables

Index of Listings

List of Acronyms and Notations

Acronyms	Meanings
ADB	Android Debug Bridge, a tool to connect to an Android emulator of smartphone to run commands, start applications, send events and perform other operations for testing and debuging purpose
AOSP	Android Open Source Project, the project hosting the most of the Android operating system source code
API	Application Programming Interface, in the Android ecosystem, it is a set of classes with known method signatures that can be called by an application to interact with the Android framework
APK	Android Package, the file format used to install application on Android. The APK format is an extention of the JAR format
ART	Android RunTime, the runtime environement that execute an Android application. The ART is the successor of the older Dalvik Virtual Machine
AXML	Android XML. The specific flavor of XML used by Android. The main specificity of AXML is that it can be compile in a binary version inside an APK
CFG	Control-Flow Graph, a graph representing the control structures (*e.g.* "if" blocks) of the code of a method/application
DEX	Dalvik Executable, the file format for the bytecode used for applicatiobs by Android
DFG	Data-Flow Graph, a graph representing the flow of information in an application
FR	Finishing Rate, the number of runs that finished over the number of total runs of an analysis
HPC	High-Performance Computing, the use of supercomputers and computer clusers
MWE	Minimum Working Example, in this context, a small example that can be used to check if a tool is working
IDE	Integrated Development Environment, a software providing tools for software development
JAR	Java ARchive file, the file format used to store several java class files. Sometimes used by Android to store DEX files instead of java classes

Acronyms	Meanings
JNI	Java Native Interface, the native library used to interact with Java classes of the application and Android API
OAT	Of Ahead Time, an ahead of time compiled format for DEX files
NDK	Native Development Kit, the set of tools used to build C and C++ code for Android
SDK	Software Development Kit, a set of tools for developing software targeting a specific platform. In the context of Android, the version of the SDK can be associated to a version of Android, and application compatibility is defined in term of compatible SDK version
XML	eXtensible Markup Language, a language to store data
ZIP	ZIP is an archive format. A ZIP file contains other files, that may be compressed

Chapter 1

Introduction

If during the next 12 months any one of you says "but we have always done it that way", I will instantly materialize beside you and I will haunt you for 24 hours.

— Rear Admiral Grace Hopper

Android is the most used mobile operating system since 2014, and since 2017, it even surpasses Windows all platforms combined[1]. The public adoption of Android is confirmed by application developers, with 1.3 million apps available in the Google Play Store in 2014, and 3.5 million apps available in 2017[2]. Its popularity makes Android a prime target for malware developers. Indeed, various applications have been shown to behave maliciously, from stealing personal informations [57] to hijacking the smartphone's computing resources to mine cryptocurrency [2].

Considering the importance of Android in the everyday life of so many people, Google, the company that develops Android, defined a very strong security model that addresses an extensive threat model [43]. This threat model goes as far as to consider that an adversary can have physical access to an unlocked device (*e.g.*, an abusive partner, or a border control). On the device, this security model includes the sandboxing of each application, controlled using a system of permissions to allow the applications to perform potentially unwanted actions. For example, an application cannot access the contact list without requesting permission from the user first. Android keeps improving its security from version to version by improving the sandboxing (*e.g.*, starting with Android 10, applications can no longer access the clipboard if they are not focused) or by using safer defaults (*e.g.*, since Android 9, by default, all network connections must use TLS).

In the spirit of *defence in depth*, Google developed a *Bouncer* service that scans applications in

1. https://gs.statcounter.com/os-market-share#monthly-200901-202304
2. https://www.statista.com/statistics/266210

the store for malicious software[1]. Although its operation is kept secret, it seems that the Bouncer is both comparing the applications with known malware code and running the applications in Google's cloud infrastructure to detect hidden behaviour. Despite Google's efforts, malicious applications are still found in the Play Store [2]. Also, it is not uncommon for people in abusive situations to have their abuser install on their phone a stalkerware (spying application) found outside of the Play Store [20].

For these reasons, it is important to be able to analyse an application and understand what it does. This process is called reverse engineering. A lot of work has been done to reverse engineer computer software, but Android applications come with specific challenges that need to be addressed. For instance, Android applications are distributed in a specific file format, the APK format, and the code of the application is mainly compiled into an Android-specific bytecode: Dalvik. An Android reverse engineer will need tools that can read those Android-specific formats. A first test in the process of reverse engineering an application would be to simply read the content of the application and the code in it. Tools like Apktool can be used to convert the binary files of an application into a human-readable format. Other tools like Jadx can go further and try to generate Java code from the bytecode in the application. Because Android applications tend to be quite large, it can be quite tedious to understand what it does just from reading their bytecode. To address this issue, many tools/approaches have been developed [33, 61] to extract higher-level information about the behaviour of the application without having to manually analyse the application. For example, Flowdroid [8] aims to detect information leaks: given a set of methods that can generate private information, and a set of methods that send information to the outside, Flowdroid will detect if private information is sent to the outside. Once again, those kinds of tools need to target Android specifically. Android runs its applications code differently than a computer would run software. One example would be the handling of entry points: computer software usually has one entry point, whereas Android applications have many, and Android will choose depending on context. Unfortunately, those tools are hard to use, and even when they work on small example applications, it is not uncommon for them to fail to run on real-life applications [53]. This is worrying. Android applications are becoming more complex every year, and tools that cannot handle this complexity will fail more often. This leads us to our first problem statement:

1. https://googlemobile.blogspot.com/2012/02/android-and-security.html

Pb1: *To what extent are previously published Android analysis tools still usable today, and what factors impact their reusability?*

Many tools have been published to analyse Android applications, but the Android ecosystem is evolving rapidly. Tools developed 5 years ago might not be usable anymore. We will endeavour to identify which tools are still usable today, and for the others, what causes them to no longer be an option.

Another issue is that Android application developers sometimes use various techniques to slow down reverse engineering. This process is called obfuscation. Malware developers do that to hide malicious behaviour and avoid detection, but the use of obfuscation is no proof that an application is malicious. Indeed, legitimate application developers can also use obfuscation to protect their intellectual property. Thus, developers and reverse engineers are playing a game of cat and mouse, constantly inventing new techniques to hide or reveal the behaviour of an application.

There are two types of reverse engineering techniques: static and dynamic. Static analysis consists of examining the application without running it, while dynamic analysis studies the action of the application while it is running. Both methods have their drawbacks, and techniques will often capitalise on the drawbacks of one of those methods. For instance, an application can try to detect if it is running in a sandbox environment and not act maliciously if it is the case. Similarly, an application can dynamically load bytecode at runtime, and this bytecode will not be available during a static analysis. Dynamic code loading relies on Java classes called `ClassLoader` that are central components of the Android runtime environment. Because dynamic code loading is such a difficult problem for static analysis, dynamic class loading is often ignored when doing static analysis. However, class loading is not limited to dynamic code loading. As a matter of fact, the Android Runtime is constantly performing class loading to load classes from the application or from the Android platform itself. This blind spot in static analysis tools raises our second problem statement:

Pb2: *What is the default Android class loading algorithm, and does it impact static analysis?*

Class loading is an operation often ignored by static analysis tools. The exact algorithm used is not well known and might not be accurately modelled by static analysis tools. If it is the case, discrepancies between the model of the tools and the one used by Android could be used as a base for new obfuscation techniques.

Reflection is another common obfuscation technique against static analysis. Instead of directly invoking methods, the generic `Method.invoke()` API is used, and the method is retrieved from

its name in the form of a character string. Finding the value of this string can be quite difficult to determine statically, so it is once again an issue more suitable for dynamic analysis. When encountering a complex case of reflection (*i.e.*, using ciphered strings) or code loading, a reverse engineer will switch to dynamic analysis to collect the relevant data (the name of the methods called or the code that was loaded), then switch back to static analysis. This is doable for a manual analysis; unfortunately, the more complex tools that would require that runtime information to perform an accurate analysis may not have a way to access this new data. Some contributions made the results they computed available to other tools by modifying the application (instrumenting) in a way that reflects those results. This led us to our last problem statement:

Pb3: *Can we use instrumentation to provide dynamic code loading and reflection data collected dynamically to static analysis tools and improve their results?*

Dynamic code loading and reflection are problems most suited for dynamic analysis. However, static analysis tools do not have access to collected data. Encoding this information inside valid applications could be a way to make it universally available to any static analysis tool. Ideally, this encoding should not degrade the quality of the static analysis compared to the original application.

Contributions

The contributions of this thesis are the following:

1. We evaluate the reusability of Android static analysis tools published by the community: we rebuild static analysis tools in their original environment as container images. With those containers, those tools are now readily available on any environment capable of running either Docker or Singularity. We tested those tools on a dataset of real-life applications balanced in order to have a significant number of applications with different characteristics to assess which characteristics impact the success of a tool. This work was presented at the 21st International Conference on Software and Systems Reuse (ICSR 2024) conference [44].
2. We model the default class loading behaviour of Android. Based on this model, we define a class of obfuscation techniques that we call *shadow attacks* where a class definition in an APK shadows the actual class definition. We show that common state-of-the-art tools like Jadx or Flowdroid do not implement this model correctly and thus can fall for those shadow attacks. We analysed a large number of recent Android applications and found that applications with class shadowing do exist, though they are the result of quirks in the APK compilation process and not deliberate obfuscation attempts. This work was published in the Digital Threats journal [45].

3. We propose an approach to allow static analysis tools to analyse applications that perform dynamic code loading: We collect at runtime the bytecode dynamically loaded and the reflection calls information, and patch the APK file to perform those operations statically. Finally, we evaluate the impact this transformation has on the tools we containerised previously.
4. We released under the GPL licence the software we used in the experiments presented in this thesis. For Chapter 3, this includes the code used to test the output of each tool and the code to analyse the results of the experiment, in addition to the containers to run the tested tools. We also released Androscalpel, a Rust crate to manipulate Dalvik bytecode, that we used to create Theseus, a set of scripts that implement the approach presented in Chapter 5. The complete list and location of the software we release are available in Appendix A.

Outline

This dissertation is composed of 6 chapters. This introduction is the first chapter. It is followed by Chapter 2, which gives background information about Android and the various analysis techniques targeting Android applications.

The next three chapters are dedicated to the contributions of this thesis. First Chapter 3 studies the reusability of static analysis tools. Next, in Chapter 4, we model the default class loading algorithm used by Android and show the consequences for reverse engineering tools that implement the wrong model. Then Chapter 5 presents an approach that allows for static analysis tools to analyse applications that load bytecode at runtime.

Finally, Chapter 6 summarises the contributions of this thesis and opens perspectives for future work.

Chapter 2

Background and Motivation

This is a Unix system. I know this.

— Alexis "Lex" Murphy, Jurassic Park

2.1 Introduction

In order to understand the challenges of reverse engineering Android applications, we first need to understand some key concepts and specificities of Android. In particular, the format in which applications are distributed, as well as the runtime environment that runs those applications, are very specific to Android. To handle those specificities, a reverse engineer must have appropriate tools. Some of those tools are used recurrently, either by the reverse engineer themself, or as a basis for other more complex tools that implement more advanced analysis techniques.

Among those techniques, the ones that do not require running the application are called static analysis. Over time, many of those tools have been released. To compare those different tools, different benchmarks have been proposed, highlighting different strengths and weaknesses of each tool.

Unfortunately, static analysis has its limits. One such limit is that it cannot analyse what is not inside the application. Platform classes are classes that are present directly on the smartphone, and not in the application. Some of those classes are well known and taken into account by analysis tools, but the rest of those classes, often called *hidden API*, are not. In addition to platform classes, classes that are loaded dynamically (*i.e.*, at runtime) are also not always available to static analysis. This led static analysis tools to disregard the class loading process altogether, leaving the subject relatively unexplored.

When static analysis fails, for instance, because of dynamic class loading, the reverse engineer will fall back on dynamic analysis. Dynamic analysis is the counterpart of static analysis: it is based on the analysis of the execution of the application. Depending on the context, the reverse engineer will then alternate between different techniques, using previous results to improve the

next iteration. Regrettably, analysis tools mostly return results in an ad hoc format, making it difficult to make other tools aware of the retrieved information. Some tools, however, encode their result in the form of a new augmented Android application. The idea being that any Android analysis tools must be able to handle an Android application in the first place, so they will have access to that new information.

We will begin this chapter with a presentation of the bases of the Android ecosystem. The reader already familiar with Android reverse engineering might want to skip Section 2.2 and directly read Section 2.3, where we put our problem statements in perspective. We will then examine the state of the art related to those problem statements in Section 2.4, and conclude this chapter in Section 2.5.

2.2 Android Background

We begin this chapter with background information about Android and the reverse engineering of Android applications. We start with a description of Android applications and their execution environment, then list some useful basic tools for reverse engineering, and finish with the basics of static analysis for Android.

2.2.1 Android

Android is the smartphone operating system developed by Google. It is based on a Long Term Support Linux Kernel, to which patches developed by the Android community are added. On top of the kernel, Android redeveloped many of the usual components used by Linux-based operating systems, like the init system or the standard C library, and added new ones, like the ART that executes the applications. Those changes make Android a unique operating system.

2.2.1.1 Android Applications

Applications in the Android ecosystem are distributed in the APK format. APK files are JAR files with additional features, which are themselves ZIP files with additional features.

A minimal APK file contains a file `AndroidManifest.xml`, the `META-INF/` folder containing the JAR manifest and signature files, and an APK Signing Block at the end of the ZIP file. The code of the application is then stored in a custom format, the Dalvik bytecode, or in the binary ELF format, called native code in the Android ecosystem, or both. Dalvik bytecode is stored in the `classes.dex`, `classes2.dex`, `classes3.dex`, … while native code is stored in `lib/<arch>/*.so`. The `res/` folder contains the resources required for the user interface. When resources are present in `res/`, the file `resources.arsc` is also present at the root of the archive. The `assets/` folder contains the files that are used directly by the code application. Depending on the application and compilation process, any kind of other files and folders can be added to the application.

Signature Android applications are cryptographically signed to prove the authorship. Applications signed with the same key are considered developed by the same entity. This allows updating the applications securely, and applications can declare security permissions to restrict access to some features to only applications with the same author.

Android has several signature schemes coexisting:
- The v1 signature scheme is the JAR signing scheme, where the signature data is stored in the `META-INF/` folder.
- The v2, v3 and v3.1 signature scheme are store in the 'APK Signing Block' of the APK. The v2 signature scheme was introduced in Android 7.0, and to keep retro-compatibility with older versions, the v1 scheme is still used in addition to the APK Signing Block. The Signing block is an unindexed binary section added to the ZIP file, between the ZIP entries and the Central Directory. The signature was added in an unindexed section of the ZIP to avoid interfering with the v1 signature scheme that signed the files inside the archive, and not the archive itself.
- The v4 signature scheme is complementary to the v2/v3 signature scheme. Signature data are stored in an external, `.apk.idsig` file.

Android Manifest The Android Manifest is stored in the `AndroidManifest.xml`, encoded in the binary AXML format. The manifest declares important information about the application:
- Generic information like the application name, ID and icon.
- The Android compatibility of the applications, in the form of 3 values: the Android `min-sdk`, `target-sdk` and `max-sdk`. Those are the minimum, targeted and maximum versions of the Android SDK supported by the application.
- The application components (Activity, Service, Receiver and Provider) of the application and their associated classes.
- Intent filters to list the intents that can start or be sent to the application components.
- Security permissions required by the application.

Code An application usually contains at least a `classes.dex` file containing Dalvik bytecode. This is the format executed by the Android ART. It is common for an application to have more than one DEX file when an application needs to reference more methods than the format allows in one file (each method referenced inside a DEX is associated with a 16-bits number, limiting their number to 65 536). Support for multiple DEX files was added in the SDK 21 version of Android, and applications that have multiple DEX files are sometimes referred to as 'multi-dex'.

In addition to DEX files, and sometimes instead of DEX files, applications can contain `.so` ELF (Executable and Linkable Format) files in the `lib/` folder. In the Android ecosystem, binary code is called native code. Because native code is compiled for a specific architecture, `.so`

files are present in different versions, stored in different subfolders, depending on the targeted architecture. For example, `lib/arm64-v8a/libexample.so` is the version of the `example` library compiled for an ARM 64 architecture. Because smartphones mostly use ARM processors, it is not rare to see applications that only have the ARM version of their native code.

Resources Developing graphical interfaces for applications requires many kinds of specific assets, which are stored in `lib/`. Those resources include bitmap images, text, layout, etc. Data like layout, colour or text are stored in binary AXML. An additional file, `resources.arsc`, in a custom binary format, contains a list of the resource names, ids, and their properties.

Compilation Process For the developer, the compilation process is handled by Android Studio and is mostly transparent. Behind the scenes, Android Studio relies on Gradle to orchestrate the different compilation steps:

The sources XML files like `AndroidManifest.xml` and the one in `res/` are compiled to binary AXML by `aapt`, which also generates the resource table `resources.arsc` and a `R.java` file that defines for each resource variables named after the resource, set to the ID of the resource. The `R.java` file allows the developer to refer to resources with readable names and avoid using the often automatically generated resource IDs, which can change from one version of the application to another.

The source code is then compiled. The most common programming languages used for Android applications are Java and Kotlin. Both are first compiled to Java bytecode in `.class` files using the language compiler. To allow access to the Android API, the `.class` are linked during the compilation to an `android.jar` file that contains classes with the same signatures as the ones in the Android API for the targeted SDK. The `.class` files are then converted into the DEX format using `d8`. During those steps, both the original language compiler and `d8` can perform optimisations on the classes, like code shrinking, inlining, etc.

If the application contains native code, the original C or C++ code is compiled using Android tools from the NDK to target the different possible architectures.

`aapt` is then used once again to package all the generated AXML, DEX, `.so` files, as well as the other resource files, assets, `resources.arsc`, and any additional files deemed necessary to form the final ZIP file. `aapt` ensures that the generated ZIP is compatible with the requirements of Android. For instance, the `resources.arsc` will be mapped directly in memory at runtime, so it must not be compressed inside the ZIP file.

If necessary, the ZIP file is then aligned using `zipalign`. Again, this is to ensure compatibility with Android optimisations: some files like `resources.arsc` need to be 4-bits aligned to be mapped in memory.

The last step is to sign the application using the `apksigner` utility.

Since 2021, Google has required that new applications in the Google Play app store be uploaded in a new format called Android App Bundles. The main difference is that Google will perform the last packaging steps and generate (and sign) the application itself. This allows Google to generate different applications for different targets and to avoid including unnecessary files in the application, like native code targeting the wrong architecture.

2.2.1.2 Android Runtime

Android runtime environment has many specificities that set it apart from other platforms. A heavy emphasis is put on isolating the applications from one another as well as from the system's critical capabilities. The code execution itself can be confusing at first. Instead of the usual linear model with a single entry point, applications have many entry points that are called by the Android framework in accordance with external events.

Application Architecture Android application expose their components to the Android Runtime (ART) via classes inheriting specific classes from the Android SDK. Four classes represent application components that can be used as entry points:

- *Activities*: An activity represents a single screen with a user interface. This is the component used to interact with a user.
- *Services*: A service serves as an entry point to run the application in the background.
- *Broadcast receivers*: A broadcast receiver is an entry point used when a matching event is broadcast by the system.
- *Content providers*: A content provider is a component that manages data accessible by other applications through the content provider.

Components must be listed in the `AndroidManifest.xml` of the application so that the system knows of them. In the life cycle of a component, the system will call specific methods defined by the classes associated with each component type. Those methods are to be overridden by the classes defined in the application if they are specific actions to be performed. For instance, an activity might compute some values in `onCreate()`, called when the activity is created, save the value of those variable to the file system in `onStop()`, called when the acitivity stop being visible to the user, and recover the saved values in `onRestart()`, called when the user navigate back to the activity.

In addition to the components declared in the manifest that act as entry points, the Android API heavily relies on callbacks. The most obvious cases are for the user interface; for example, a button will call a callback method defined by the application when clicked. Other parts of the API also rely on non-linear execution; for example, when an application sends an intent (see next paragraph), the intent sent in response is transmitted back to the application by calling another method.

Application Isolation and Interprocess Communication On Android, each application has its own storage folders and the application processes are isolated from each other, and from the hardware interfaces. This sandboxing is done using Linux security features like group and user permissions, SELinux, and seccomp. The sandboxing is adjusted according to the permissions requested in the `AndroidManifest.xml` file of the applications. In addition, most features of the Android system can only be accessed through Binder, Android's main interprocess communication channel.

Binder is a component of the Android framework, external to the application, that all applications can communicate with. Applications can send messages to Binder, called **intents**. Binder will check if the application is allowed to send it, and then forward it to the appropriate component. This component can then respond with another intent. Applications must declare intent filters to indicate which intent can be sent to the application, and which classes receive the intents. Intents are central to Android applications and are not just used to access Android capabilities. For instance, activities and services are started by receiving intents, and it is not uncommon for an application to self-send intents to switch between activities. Intents can also be sent directly from Android to the application: when a user starts an application by tapping the app icon, Android will send an intent to the class of the application that defined the intent filter for the `android.intent.action.MAIN` intent. One interesting feature of the Binder is that intents do not need to explicitly name the targeted application and class: intents can be implicit and request an action without knowing the exact application that will perform it. An example of this behaviour is when an application wants to open a file: an `android.intent.action.VIEW` intent is sent with the file location and type, and Binder will find and start an application capable of viewing this file.

Platform Classes In addition to the classes they include, Android applications have access to classes provided by Android, stored on the phone. Those classes are called *platform classes*. They are divided between SDK classes and hidden API. The SDK classes can be seen as the Android standard library. They are documented by Google and have a certain stability from version to version. In case of breaking changes, the changes are listed by Google as well. The list of SDK classes is available at compile time in the form of an `android.jar` file to link against.

On the contrary, hidden API are undocumented methods used internally by the ART. Still, they are loaded by the application and can be used by it.

Class Loading and Reflection Class loading is the mechanism used by Android to find and select the class implementation when encountering a reference to a class. Android developers mainly use it to load bytecode dynamically from a source other than the application itself (*e.g.*, a file downloaded at runtime), using `ClassLoader` objects. `Class` objects are retrieved from those class loaders using their name in the form of strings to identify them. Those `Class` can then

be instantiated into an object, and `Methods` objects can be used to call the methods of the instantiated object. The process of manipulating `Class` and `Methods` objects instead of using bytecode instructions is called reflection. Reflection is not limited to bytecode that has been dynamically loaded: it can be used for any class or method available to the application.

Because the `ClassLoader` objects are only used when loading bytecode dynamically or when using reflection, it is often forgotten that the ART uses class loaders constantly behind the scene, allowing classes from the application and platform classes to cohabit seamlessly.

∴

In this subsection, we presented the most notable specificities of the Android ecosystem. In the next section, we will continue with the various tools available for an Android reverse engineer.

2.2.2 Reverse Engineering Tools

Due to the specificities of Android, reverse engineers need tools adapted to Android. The development tools provided by Google can be used for basic operations, but a reverse engineer will quickly need more specialised tools. Usually, the first step while analysing an application is to look at its content. Apktool and Jadx are common tools used to convert the content of an application into a readable format. Analysing an application this way, without running it, is called static analysis. For more advanced forms of static analysis, Androguard and Soot can be used as libraries to automate analyses. When static analysis becomes too complicated (*e.g.*, if the application uses obfuscation techniques), a reverse engineer might switch to dynamic analysis. This time, the application is executed, and the analyst will scrutinise the behaviour of the application. Frida is a good option to help with this dynamic analysis. It is a toolkit that can be used to intercept method calls and execute custom scripts while an application is running.

Android Studio The whole Android development ecosystem is packaged by Google in the IDE Android Studio[1]. In practice, Android Studio is a source-code editor that wraps around the different tools of the Android SDK. The SDK tools and packages can be installed manually with the `sdkmanager` tool. Among the notable tools in the SDK are:

- `emulator`: this tool allows running an emulated Android phone on a computer. Although very useful, Android emulator has several limitations. For once, it cannot emulate another architecture. An x86_64 computer cannot emulate an ARM smartphone. This can be an issue because a majority of smartphones run on ARM processors. Also, for certain versions of Android, the proprietary GooglePlay libraries are not available on rooted emulators. Lastly, emulators are not designed to be stealthy and can easily be detected by an application. Malware will avoid detection by not running its payload on emulators.

1. https://developer.android.com/studio

- ADB: a tool to send commands to an Android smartphone or emulator. It can be used to install applications, send instructions, events, and generally perform debugging operations.
- Platform Packages: Those packages contain data associated with a version of Android needed to compile an application. Especially, they contain the so-called `android.jar` files, which contain the list of API for a version of Android.
- `d8`: The main use of `d8` is to convert Java bytecode files (`.class`) to Android DEX format. It can also be used to perform different levels of optimisation of the bytecode generated.
- `aapt/aapt2` (Android Asset Packaging Tool): This tool is used to build the APK file. It is commonly used by other tools that repackage applications like Apktool. Behind the scenes, it converts XML to binary AXML and ensures that each file has the right compression and alignment. (*e.g.*, some resource files are mapped in memory by the ART, and thus need to be aligned and not compressed).
- `apksigner`: the tool used to sign an APK file. When repackaging an application, for example, with Apktool, the new application needs to be signed.

Apktool Apktool[1] is a *reengineering tool* for Android APK files. It can be used to disassemble an application: it will extract the files from the APK file, convert the binary AXML to text XML, and use smali/backsmali[2] to convert the DEX files to smali, an assembler-like language that matches the Dalvik bytecode instructions. The main strength of Apktool is that after disassembling an application, its content can be edited and reassembled into a new APK.

Androguard Androguard[3] [15] is a Python library for parsing and disassembling APK files. It can be used to automatically read Android manifests, resources, and bytecode. Contrary to Apktool, which generates text files, it can be used as a library to programmatically analyse the application. It can also perform additional analysis, like computing a call graph or control flow graph of the application (we will explain what those graphs are later in Section 2.2.3). However, contrary to Apktool, it cannot repackage a modified application.

Jadx Jadx[4] is an application decompiler. It converts DEX files to Java source code. It is not always capable of decompiling all classes of an application, so it cannot be used to recompile a new application, but the code generated can be very helpful to reverse an application. In addition to decompiling DEX files, Jadx can also decode Android manifests and application resources.

Soot Soot[5] [7] was originally a Java optimisation framework. It could lift Java bytecode to other intermediate representations that can be optimised, then converted back to bytecode.

1. https://apktool.org/
2. https://github.com/JesusFreke/smali
3. https://github.com/androguard/androguard
4. https://github.com/skylot/jadx
5. https://github.com/soot-oss/soot

Because Dalvik bytecode and Java bytecode are equivalent, support for Android was added to Soot, and Soot features are now leveraged to analyse and modify Android applications. One of the best-known examples of Soot usage for Android analysis is Flowdroid [8], a tool that computes data flow in an application.

A new version of Soot, SootUp[1], is currently being worked on. Compared to Soot, it has a modernised interface and architecture, but it is not yet feature-complete, and some tools like Flowdroid are still using Soot.

Frida Frida[2] is a dynamic instrumentation toolkit. It allows the reverse engineer to inject and run JavaScript code inside a running application.

To instrument an application, the Frida server must be running as root on the phone, or the Frida library must be injected inside the APK file before installing it. Frida defines a JavaScript wrapper around the Java Native Interface (JNI) used by native code to interact with Java classes and the Android API. In addition to allowing interaction with Java objects from the application and the Android API, this wrapper provides the option to replace a method implementation with a JavaScript function (that itself can call the original method implementation if needed). This makes Frida a powerful tool capable of collecting runtime information or modifying the behaviour of an application as needed.

The main drawback of using Frida is that it is a known tool, easily detected by applications. Malware might implement countermeasures that avoid running malicious payloads if Frida is detected.

∴

Those tools are quite useful for manual operations. However, considering the complexity of modern Android applications, it might take a lot of work for a reverse engineer to analyse one application. Different techniques have been developed to streamline the analysis. Next, we will see the most common of those techniques for static analysis.

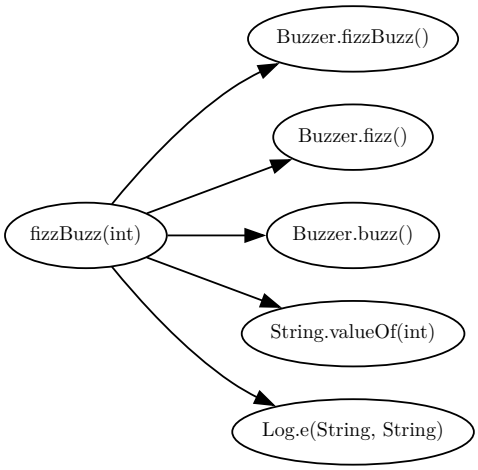
2.2.3 Static Analysis

A static analysis program examines an APK file without executing it to extract information from it. Basic static analysis can include extracting information from the `AndroidManifest.xml` file or decompiling bytecode to Java code with tools like Apktool or Jadx. Unfortunately, simply reading the bytecode does not scale. To do so, a human analyst is needed, making it complicated to analyse a large number of applications, and even for single applications, the size and complexity of some applications can quickly overwhelm the reverse engineer.

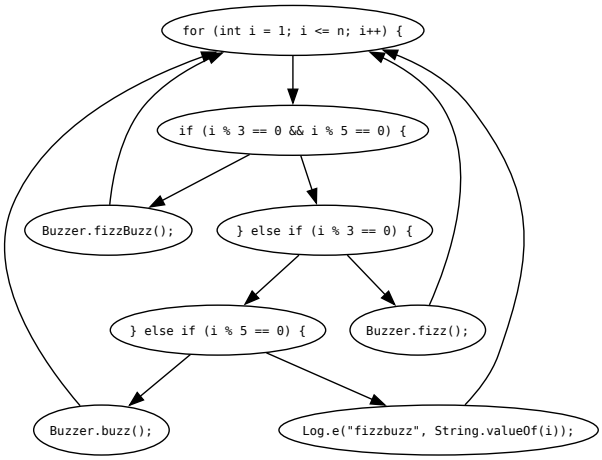
1. https://github.com/soot-oss/SootUp
2. https://frida.re/

```java
public static void fizzBuzz(int n) {
  for (int i = 1; i <= n; i++) {
    if (i % 3 == 0 && i % 5 == 0) {
      Buzzer.fizzBuzz();
    } else if (i % 3 == 0) {
      Buzzer.fizz();
    } else if (i % 5 == 0) {
      Buzzer.buzz();
    } else {
      Log.e("fizzbuzz", String.valueOf(i));
    }
  }
}
```

a) A Java program



b) Corresponding Call Graph



c) Corresponding Control-Flow Graph

Figure 1: Source code for a simple Java method and its Call and Control Flow Graphs

Control flow analysis is often used to mitigate this issue. The idea is to extract the behaviour, the flow, of the application from the bytecode, and to represent it as a graph. A graph representation is easier to work with than a list of instructions and can be used for further analysis. Depending on the level of precision required, different types of graphs can be computed. The most basic of those graphs is the call graph. A call graph is a graph where the nodes represent the methods in

the application, and the edges represent calls from one method to another. Figure 1 b) show the call graph of the code in Figure 1 a). A more advanced control-flow analysis consists of building the control-flow graph. This time, instead of methods, the nodes represent instructions, and the edges indicate which instruction can follow which instruction. Figure 1 c) represents the control-flow graph of Figure 1 a), with code statements instead of bytecode instructions.

Once the control-flow graph is computed, it can be used to compute data-flows. Data-flow analysisis used to follow the flow of information in the application. By defining a list of methods and fields that can generate critical information (taint sources) and a list of methods that can consume information (taint sinks), taint-tracking detects potential data leaks (if a data flow links a taint source and a taint sink). For example, `TelephonyManager.getImei()` returns a unique, persistent, device identifier. This can be used to identify the user, and it cannot be changed if compromised. This makes `TelephonyManager.getImei()` a good candidate as a taint source. On the other hand, `UrlRequest.start()` sends a request to an external server, making it a taint sink. If a data-flow is found linking `TelephonyManager.getImei()` to `UrlRequest.start()`, this means the application is potentially leaking critical information to an external entity, a behaviour that is probably not wanted by the user.

Static analysis is powerful as it can detect unwanted behaviour in an application, even if the behaviour does not manifest itself when running the application. However, static analysis tools must overcome many challenges when analysing Android applications.

the Java object-oriented paradigm A call to a method can, in fact, correspond to a call to any method overriding the original method in subclasses.

the multiplicity of entry points Each component of an application can be an entry point for the application.

the event-driven architecture Methods in the applications can be called when events occur, in an unknown order.

the interleaving of native code and bytecode Native code can be called from bytecode and vice versa, but tools often only handle one of those formats.

the potential dynamic code loading An application can run code that was not originally in the application.

the use of reflection Methods can be called from their name as a string object, which is difficult to identify statically.

the continual evolution of Android each new version of Android brings new features that analysis tools must be aware of. For instance, the multi-dex feature presented in Section 2.2.1.1 was introduced in Android SDK 21. Tools unaware of this feature only analyse the `classes.dex` file and will ignore all other `classes<n>.dex` files.

∴

With the bases of Android application analysis in mind, we can now examine our problem statements further.

2.3 Problems of the Reverse Engineer

In this section, we will develop on some issues encountered by reverse engineers, and link them to our problem statements.

In the previous section, we listed some limitations to static analysis. Some limitations have been known for some time now, and many contributions have been made to overcome them. Those contributions often introduce new tools that implement solutions to those different issues. Depending on the situation, a reverse engineer might want to use those tools or build another tool on top of one. Unfortunately, they can be hard to use. And like we said previously, the fast evolution of Android can be a significant obstacle. The combination of those two points can lead a reverse engineer to spend a lot of time trying to use a tool without realising that the tool does not work anymore. Our first problem statement **Pb1** focuses on this issue: *To what extent are previously published Android analysis tools still usable today, and what factors impact their reusability?* Determining which tools are still usable today is a first step, but finding out what reasons make a tool stop working might help write more resilient tools in the future.

We also presented dynamic code loading as an obstacle for static analysis. Code loading is achieved using class loader objects, causing class loaders to be generally associated with dynamic code loading. However, class loading plays a much more important role in the ART. Class loading originates from the Java ecosystem and was ported to Android so that developers could keep writing applications in Java. Despite that, Android made a lot of changes to the original Java classes and did not document those changes. Between static analysis, general oversight of class loading, relegating it to dynamic analysis, and the lake of documentation of the actual behaviour of the ART, the question of the impact of the class loading algorithm on static analysis can be asked. Our second problem statement **Pb2** aims to anwser this question: *What is the default Android class loading algorithm, and does it impact static analysis?*

Circling back to known limitations of static analysis, dynamic code loading and reflection are often used to obfuscate applications. Dynamic code loading allows hiding bytecode from static analysis with relatively low effort. The bytecode can be downloaded at runtime, stored in the application encrypted, hidden inside other files, generated at runtime, etc. In a way, reflection can do the same thing, but for specific method calls: instead of the actual call, static analysis will see a call to the generic `Method.invoke()` method. By contrast, it is relatively easy to find the name of the method called or to intercept dynamically loaded bytecode using dynamic tools like Frida. The issue that arises then is what to do with the collected data. Simply having it greatly helps a manual analysis, but it cannot be used directly by tools that perform static analyses. There is no standard representation for runtime information, and there is simply no

way to give a list of reflection sites and the associated method calls as a new input for most static analysis tools. This means that in most cases, when a reverse engineer wants to improve static analysis with dynamic analysis, they need to modify the static tools to receive the additional runtime data. Doing so requires both time and knowledge of the internals of the tools used. Our third problem statement, **Pb3**, explores an alternative approach that modifies the application instead of the tool: *Can we use instrumentation to provide dynamic code loading and reflection data collected dynamically to static analysis tools and improve their results?*

We will now explore the current state of the art for relevant contributions related to our problem statements.

2.4 State of the Art

This section focuses on the state of the art related to our three problem statements: the reusability of Android static analysis tools, the class loading mechanism of Android, and the use of instrumentation to encode information collected dynamically.

2.4.1 Reusability of Static Analysis Tools

To what extent are previously published Android analysis tools still usable today, and what factors impact their reusability?

In the past fifteen years, the research community has released many tools to detect or analyse malicious behaviours in applications. The first step to answer this question is to list those previously published tools. The number of publications related to static analysis can make it difficult to find the right tool for the right task. Li *et al.* [33] published a systematic literature review for Android static analysis before May 2015. They analysed 92 publications and classified them by goal, method used to solve the problem and underlying technical solution for handling the bytecode when performing the static analysis. In particular, they listed 27 approaches with an open-source implementation available.

Interestingly, a lot of the tools listed rely on common tools to interact with Android applications/DEX bytecode. Recuring examples of such support tools are Apktool (*e.g.*, Amandroid [66], Blueseal [58], SAAF [26]), Androguard (*e.g.*, Adagio [21], Appareciumn [63], Mallodroid [19]) or Soot (*e.g.*, Blueseal [58], DroidSafe [24], Flowdroid [8]): those tools are built incrementally, on top of each other. This strengthens our idea that being able to reuse previous tools is important.

Nevertheless, Li *et al.* focus more on the techniques and features described in the reviewed publications, and experiments to evaluate whether the pointed out software are still usable were not performed. We believe that the effort of reviewing the literature for making a comprehensive overview of available approaches should be pushed further: an existing published approach with

a software that cannot be used for technical reasons endangers both the reproducibility and reusability of research.

We will now explore this direction further by looking at other works that have been done to evaluate different analysis tools. Those evaluations often take the form of benchmarks and follow a similar method (we will look at the different contributions in more details in Section 2.4.1.2). They start by selecting a set of tools with similar goals to compare. Usually, those contributions are comparing existing tools to their own, but some contributions do not introduce a new tool and focus on surveying the state of the art for some technique. They then selected a dataset of applications to analyse. We will see in Section 2.4.1.1 that those datasets are often hand-crafted, except for some studies that select a few real-world applications that they manually reverse-engineered to get a ground truth to compare to the tool's result. Once the tools and test dataset are selected, the tools are run on the application dataset, and the results of the tools are compared to the expected results (ground truth) to determine the accuracy of each tool. Additional factors are sometimes compared as well: the number of false positives, false negatives, or even the time it took to finish the analysis. Occasionally, the number of applications a tool simply failed to analyse is also compared.

In Section 2.4.1.1, we will look at the dataset used in the community to compare analysis tools. Then, in Section 2.4.1.2, we will go through the contributions that benchmarked those tools to see if they can be used as an indication as to which tools can still be used today.

2.4.1.1 Application Datasets

Research contributions often rely on existing datasets or provide new ones in order to evaluate the developed software. Raw datasets such as Drebin[6] contain little information about the provided applications. As a consequence, dataset suites have been developed to provide, in addition to the applications, meta information about the expected results. For example, taint analysis datasets should provide the source and expected sink of a taint. In some cases, the datasets are provided with additional software for automating part of the analysis. One such dataset is DroidBench, which was released with the tool Flowdroid [8]. Later, the dataset ICC-Bench was introduced with the tool Amandroid [66] to complement DroidBench by introducing applications using Inter-Component data flows. These datasets contain carefully crafted applications containing flows that the tools should be able to detect. These hand-crafted applications can also be used for testing purposes or to detect any regression when the software code evolves. The drawback to using hand-crafted applications is that these datasets are not representative of real-world applications [50] and the obtained results can be misleading.

Contrary to DroidBench and ICC-Bench, some approaches use real-world applications. Bosu *et al.* [12] use DIALDroid to perform a threat analysis of Inter-Application communication and published DIALDroid-Bench, an associated dataset. Similarly, Luo *et al.* released Taint-

Bench [40], a real-world dataset, and the associated recommendations to build such a dataset. These datasets are useful for carefully spotting missing taint flows, but contain only a few dozen applications.

In addition to those datasets, AndroZoo [3] collect applications from several application marketplaces, including the Google Play store (the official Google application store), Anzhi and AppChina (two Chinese stores), or FDroid (a store dedicated to free and open source applications). Currently, Androzoo contains more than 25 million applications that can be downloaded by researchers from the SHA256 hash of the application. Androzoo also provides additional information about the applications, like the date the application was detected for the first time by Androzoo or the number of antiviruses from VirusTotal that flagged the application as malicious. This will allow us to sample a dataset of applications evenly distributed over the years. In addition to providing researchers with easy access to real-world applications, Androzoo makes it a lot easier to share datasets for reproducibility: instead of sharing hundreds of APK files, the list of SHA256 is enough.

2.4.1.2 Benchmarking

We will now go through the different contributions that evaluated different static analysis tools to see if they can give us some insights into the current usability of the tools.

The few experiments with datasets composed of real-world applications confirmed that some tools, such as Amandroid [66] and Flowdroid [8], are less efficient on real-world applications [12, 40]. Unfortunately, those real-world applications datasets are rather small, and a larger number of applications would be more suitable for our goal, *i.e.*, evaluating the reusability of a variety of static analysis tools.

Pauck *et al.* [49] used DroidBench [8], ICC-Bench [66] and DIALDroid-Bench [12] to compare Amandroid [66], DIAL-Droid [12], DidFail [28], DroidSafe [24], FlowDroid [8] and IccTA [30]. To perform their comparison, they introduced the AQL (Android App Analysis Query Language) format. AQL can be used as a common language to describe the computed taint flow as well as the expected result for the datasets. It is interesting to notice that all the tested tools timed out at least once on real-world applications, and that Amandroid [66], DidFail [28], DroidSafe [24], IccTA [30] and ApkCombiner [31] (a tool used to combine applications) all failed to run on applications built for Android API 26. These results suggest that a more thorough study of the link between application characteristics (*e.g.*, date, size) should be conducted. Luo *et al.* [40] used the framework introduced by Pauck *et al.* to compare Amandroid [66] and Flowdroid [8] on DroidBench and their own dataset TaintBench, composed of real-world Android malware. They found out that those tools have a low recall on real-world malware, and are thus over-adapted to micro-datasets. Unfortunately, because AQL is only focused on taint flows, we cannot use it to evaluate tools performing more generic analysis.

A first work about quantifying the reusability of static analysis tools was proposed by Reaves *et al.* [53]. Seven Android analysis tools (Amandroid [66], AppAudit [69], DroidSafe [24], Epicc [48], FlowDroid [8], MalloDroid [19] and TaintDroid [17]) were selected to check if they were still readily usable. For each tool, both the usability and results of the tool were evaluated by asking auditors to install and use it on DroidBench and 16 real-world applications. The auditors reported that most of the tools require a significant amount of time to set up, often due to dependency issues and operating system incompatibilities. Reaves *et al.* propose to solve these issues by distributing a Virtual Machine with a functional build of the tool in addition to the source code. Regrettably, these Virtual Machines were not made available, preventing future researchers from taking advantage of the work done by the auditors. Reaves *et al.* also report that real-world applications are more challenging to analyse, with tools having lower results, taking more time and memory to run, sometimes to the point of not being able to run the analysis. Considering it was noticed on a dataset of only 16 real-world applications, this result is worrying. A more diverse dataset would be needed to better assess the extent of the issue and give more insight into the factors impacting the performance of the tools.

Mauthe *et al.* present an interesting methodology to assess the robustness of Android decompilers [42]. They used 4 decompilers on a dataset of 40 000 applications. The error messages of the decompilers were parsed to list the methods that failed to decompile, and this information was used to estimate the main causes of failure. It was found that the failure rate is correlated to the size of the method, and that a consequent amount of failures are from third-party libraries rather than the core code of the application. They also concluded that malware are easier to entirely decompile, but have a higher failure rate, meaning that the ones that are hard to decompile are substantially harder to decompile than goodware.

∴

To summarise, Li *et al.* made a systematic literature review of static analysis for Android that listed 27 open-sourced tools. However, they did not test those tools. Reaves *et al.* did so for some of them and analysed the difficulty of using them. They raised two major concerns about the use of Android static analysis tools. First, they can be quite difficult to set up, and second, they appear to have difficulties analysing real-world applications. This is problematic for a reverse engineer, not only do they need to invest a significant amount of work to set up a tool properly, but they also do not have any guarantees that the tool will actually manage to analyse the application they are investigating.

In Chapter 3, we will try to set up the tools listed by Li *et al.* and test them on a large number of real-world applications to see which can be used today. We will also aim at identifying what characteristics of real-world applications make them harder to analyse.

2.4.2 Android Class Loading

What is the default Android class loading algorithm, and does it impact static analysis?

This subsection is mainly dedicated to class loading in Java and Android. Because we focus on the *default* class loading algorithm, we will not focus on dynamic code loading (*i.e.*, loading of additional bytecode while the application is already running). However, class loading is used, without dynamic code loading, to load classes other than the one in the application: the platform classes. In the second part of this subsection, we will look at the work that has been done related to those platform classes.

2.4.2.1 Class Loading

Class loading is a fundamental element of Java; it defines which classes are loaded from where. In Android, this is often associated with dynamic code loading, as the `ClassLoader` objects are used to load code at runtime. However, class loading also intervenes to load platform classes or classes from the application itself, and thus requires some attention when analysing an application.

Class loading mechanisms have been studied in the general context of the Java language. Gong [23] describes the JDK 1.2 class loading architecture and capabilities. One of the main advantages of class loading is the type safety property that prevents type spoofing. As explained by Liang and Bracha [35], by capturing events at runtime (new loaders, new class) and maintaining constraints on the multiple loaders and their delegation hierarchy, authors can avoid confusion when loading a spoofed class. This behaviour is now implemented in modern Java virtual machines. Later, Tazawa and Hagiya [64] proposed a formalisation of the Java Virtual Machine supporting dynamic class loading in order to ensure type safety. Those works ensure strong safety for the Java Virtual Machine, in particular when linking new classes at runtime. Although Android has a similar mechanism, the implementation is not shared with the JVM of Oracle. Additionally, our problem statement does not focus on spoofing classes at runtime, but on confusions that occur when using a static analyser used by a reverser that tries to understand the code loading process offline.

Contributions about Android class loading focus on using the capabilities of class loading to extend Android features or to prevent reverse engineering of Android applications. For instance, Zhou *et al.* [76] extend the class loading mechanism of Android to support regular Java bytecode, and Kritz and Maly [29] propose a new class loader to automatically load modules of an application without user interactions.

Regarding reverse engineering, class loading mechanisms are frequently used by packers, applications that load their actual code at runtime, for hiding all or parts of the code of an application [16]. For example, packers exploits the class loading capability of Android to load new code. They also combine the loading with code generation from ciphered assets or

code modification from native code calls [37] to increase the difficulty of recovery of the code. Because parts of the original code will be only available at runtime, deobfuscation approaches propose techniques that track DEX structures when manipulated by the application [68, 70, 74]. Those contributions interact with the class loading mechanism of Android to collect the DEX structures at the right moment.

Some classes, however, are not loaded from the application, nor dynamically loaded by the application. Those classes are platform classes, and apart from dynamic code loaded, they are the main reason class loading is needed by Android. We will now look at the literature related to them.

2.4.2.2 Platform Classes

Platform classes are divided between SDK classes that are documented, and the other classes, often referred to as hidden APIs. SDK classes are clearly listed and documented by Google, so they do not require as much attention as hidden APIs. As we said earlier, hidden API are undocumented methods that can be used by an application, thus making them a potential blind spot when analysing an application. However, not a lot of research has been done on the subject.

Li *et al.* did an empirical study of the usage and evolution of hidden API [34]. They found that hidden API are added and removed in every release of Android, and that they are used both by benign and malicious applications.

More recently, He *et al.* [25] did a systematic study of hidden service API related to security. They studied how the hidden API can be used to bypass Android security restrictions and found that although Google countermeasures are effective, they need to be implemented inside the system services and not the hidden API due to the lack of in-app privilege isolation: the framework code is in the same process as the user code, meaning any restriction in the framework can be bypassed by the user. Unfortunately, those two contributions do not explore further the consequences of the use of hidden APIs for a reverse engineer.

$\therefore$

In conclusion, class loading mechanisms have been studied carefully in the context of the Java language. However, the same cannot be said about Android, whose implementation diverges significantly from classic Java Virtual Machines. Most work done on Android focuses on extending Android capabilities using class loading, or on analysing dynamically the code loading operations of an application.

In Chapter 4, we will model the behaviour of Android when loaded classes used by an application that do not use dynamic code loading, and check if this behaviour matches the behaviour of

common analysis tools. We will also take some time to check if the state of the art related to hidden API is up to date with the current Android versions.

2.4.3 Allowing Static Analysis Tools to Analyse Obfuscated Application

Can we use instrumentation to provide dynamic code loading and reflection data collected dynamically to static analysis tools and improve their results?

Dynamic analysis of Android applications has been researched for a long time. Like static analysis, it has its own challenges, which we will explore in this subsection. After that, we will also look at contributions that sought to encode results inside the APK format or used instrumentation to improve analyses in some way.

2.4.3.1 Dynamic Analysis

Some situations, like reflection of dynamic code loading, are difficult to solve with static analysis and require a different approach: dynamic analysis. With dynamic analysis, the application is actually executed, and the reverse engineer observes its behaviour. Monitoring the behaviour can be achieved by various strategies: observing the filesystem, the display screen, the process memory, the kernel, etc. Depending on the chosen level of observation, dynamic analysis can become a serious technical challenge.

A basic example of dynamic analysis is presented by Bernardi *et al.* [10]: the logs generated by `strace` are used to list the system calls generated in response to an event to determine if an application is malicious or not.

More advanced methods are more intrusive and require modifying either the APK, the Android framework, runtime, or kernel. TaintDroid [17], for example, modifies the Dalvik Virtual Machine (the predecessor of the ART) to track the data flow of an application at runtime, while AndroBlare [4, 5] try to compute the taint flow by hooking system calls using a Linux Security Module. DexHunter [74] and AppSpear [73] also patch the Dalvik Virtual Machine/ ART, this time to collect bytecode loaded dynamically. Modifying the Android framework, runtime, or kernel is possible thanks to the Android project being open-source, but this is a delicate operation that requires revising a patch for each new version of Android. Thus, a common issue faced by tools that took this approach is that they are stuck with a specific version of Android. Some sandboxes limit this issue by using dynamic binary instrumentation, like DroidHook [14], based on the Xposed framework, or CamoDroid [18], based on Frida. This approach is a lot less stealthy than patching Android, but it is generally easier to set up and is easier to port to new Android versions.

Another known challenge when analysing an application dynamically is the code coverage: if some part of the application is not executed, it cannot be analysed. Considering that Android applications are meant to interact with a user, this can become problematic for automatic

analysis. The Monkey tool developed by Google is one of the most used solution [61]. It sends a random stream of events to the phone without tracking the state of the application. More advanced tools statically analyse the application to model in order to improve the exploration. Sapienz [41] and Stoat [60] use this technique to improve application testing. GroddDroid [1] has the same approach but detects statically suspicious sections of code to target, and will interact with the application to target those code sections.

Unfortunately, exploring the application entirely is not always possible, as some applications will try to detect if they are in a sandbox environment (*e.g.*, if they are in an emulator, or if Frida is present in memory) and will refuse to run some sections of code if this is the case. Ruggia *et al.* [55] make a list of evasion techniques. They propose a new sandbox, DroidDungeon, that, contrary to other sandboxes like DroidScope[71] or CopperDroid[62], strongly emphasises resilience against evasion mechanisms.

A common objective of dynamic analysis is to collect bytecode loaded dynamically and reflection information. Like we said earlier, DexHunter [74] and AppSpear [73] do that by instrumenting the Android Runtime. Qu *et al.* [52] developed DyDroid, a hybrid framework using dynamic analysis to intercept dynamic code loading and static analysis to determine the nature of the loaded code. They used DyDroid to make an audit of the use of dynamic code loading in applications from the Google Play store in 2016. It resulted that dynamic code loading was mostly related to mobile advertisement, and that the code loading originated from a third-party library included in the application, rather than the code of the application developer itself. Similarly, StaDynA [75] is a framework that generates a call graph statically, then uses dynamic analysis to analyse dynamic code loading and reflection calls to complete this call graph.

The issue with those approaches is that they are only compatible with their own subsequent analysis. For instance, StaDynA only provides the call graph, and cannot be used as is to improve the capacity of Flowdroid. This is unfortunate: the reverse engineer's next step will depend on the context. Not being able to reuse the result of a previous analysis with any ad hoc tools greatly limits their options. AppSpear has an interesting solution to this issue: the code it intercepts is repackaged inside a new APK file that Android analysis tools should be able to analyse. We will now explore further the contributions that take this approach to encode results inside applications.

2.4.3.2 Improving Analysis with Instrumentation

Usually, instrumentation refers to the practice of modifying the behaviour of a program to collect information during its execution. Frida is a good example of an instrumentation framework. The term can also be used more generally to describe operations that modify the application code. In this section, we will focus on the use of instrumentation that makes an application easier to analyse by other tools, instead of just collecting additional information at runtime.

In the previous section, we gave the example of AppSpear [73], which reconstructs DEX files intercepted at runtime and repackages the APK with the new code in it. DexLego [46] has a similar but a lot more aggressive technique. It targets heavily obfuscated packers that decrypt then re-encrypt the method's instructions just in time. To get the bytecode, DexLego logs each instruction executed by the ART, and reconstructs the methods, then the DEX files, from this stream of instructions. The main limitation of this technique is that it carries over the limitation of dynamic analysis to static analysis: the bytecode injected in the application is limited to the instructions executed during the dynamic analysis. Nevertheless, it is an interesting way to encode the traces of a dynamic analysis in a way that can be used by any Android analysis tool.

IccTa [30] technique is close to the idea of modifying the application to improve its analysis: it performs a first analysis to compute the potential inter-component communication of an application, then modifies the Jimple representation of this application before feeding it to Flowdroid to perform a taint analysis. Jimple is the intermediate language used by Soot, so even if IccTa does not generate a new application, this modified representation can probably be used by any tool based on the Soot framework or recompiled into a new application without too much effort. Samhi *et al.* [56] followed this direction to unify the analysis of bytecode and native code. Their tool, JuCify, uses Angr [59] to generate the call graph of the native code, and uses heuristics to encode this call graph into Jimple that can then be added to the Jimple generated by Soot from the bytecode of the application. Like IccTa, they use Flowdroid to analyse this new augmented representation of the application, but it should be usable by any analysis tools relying on Soot.

Finally, DroidRA [32] uses the COAL [47] solver to statically compute the reflection information. The reflection calls are transformed into direct calls inside the application using Soot. Using COAL makes DroidRA quite good at solving the simpler cases, where the names of classes and methods targeted by reflection are already present in the application. Those cases are quite common; being able to solve those without resorting to dynamic analysis is quite useful. On the other hand, COAL will struggle to solve cases with complex string manipulation and is simply not able to handle cases that rely on external data (*e.g.*, downloaded from the internet at runtime). Likewise, this can only access code loaded dynamically if the code was present inside the application without any kind of obfuscation (*e.g.*, a DEX file in the assets of the application can be analysed, but not if it is ciphered).

∴

Instrumenting applications to encode the result of an analysis as a unified representation has been explored before. It has been used by tools like AppSpear and DexLego to expose heavily obfuscated bytecode collected dynamically. Similarly, DroidRA compute reflection information statically and injects the actual method calls inside the application it returns.

However, AppSpear and DexLego focus primarily on specific obfuscation techniques, making their implementation difficult to port to more recent versions of Android, and DroidRA suffers from the limitation of static analysis. We believe that instrumentation is a promising approach to encoding that information. Especially, we think that it could be used to provide dynamic information that is not available to static analysis tools like DroidRA.

In Chapter 5, we will try to use instrumentation to combine dynamic analysis (to collect code loaded dynamically and reflection information) with static analysis, regardless of the static analysis tool used.

2.5 Conclusion

This chapter presented the specificities of Android and the usual tools used as a basis for reverse engineering applications. Many contributions have been made to static analysis, and benchmarks have been proposed to compare the different tools that resulted from those contributions. Those benchmarks raised questions about the reusability of those tools and their capacity to handle real-world applications. We then looked at platform classes and class loading, a commonly recognised limitation of static analysis. Because of that, the issue is generally relegated to dynamic analysis, leaving the details of the class loading mechanisms of Android unexplored. To complement static analysis, we continued by looking at dynamic analysis. A variety of approaches have been proposed, balancing ease of use, maintainability and stealthiness. The results of those analyses are often in an ad hoc format, making it difficult to reuse with other tools. A few exceptions, as well as some static analysis tools, proposed an interesting solution to this issue: instrumenting the analysed application to encode the results of the analysis in the form of a valid APK, a format any Android analysis tools should be able to read. We liked this solution and believe it should be studied further. This process led us to explore three problem statements:

Pb1 *To what extent are previously published Android analysis tools still usable today, and what factors impact their reusability?*

Pb2 *What is the default Android class loading algorithm, and does it impact static analysis?*

Pb3 *Can we use instrumentation to provide dynamic code loading and reflection data collected dynamically to static analysis tools and improve their results?*

In the next chapters, we will endeavour to contribute to the Android reverse engineering field by answering them.

Chapter 3

Evaluating the Reusability of Android Static Analysis Tools

I keep going back in time.

— Max Caulfield, Life is Strange "Out of Time"

This chapter intends to explore the robustness of past software dedicated to static analysis of Android applications. We pursue the community effort that identified software supporting publications that perform static analysis of mobile applications, and we propose a method for evaluating the reliability of this software. We extensively evaluate static analysis tools on a recent dataset of Android applications, including goodware and malware, that we designed to measure the influence of parameters such as the date and size of applications. Our results show that 54.55% of the evaluated tools are no longer usable and that the size of the bytecode and the min SDK version have the greatest influence on the reliability of the tested tools.

3.1 Introduction

In this chapter, we study the reusability of open source static analysis tools that appeared between 2011 and 2017, on a recent Android dataset. The scope of our study is **not** to quantify if the output results are accurate to ensure reproducibility, because all the studied static analysis tools have different goals in the end. On the contrary, we take the hypothesis that the provided tools compute the intended result, but may crash or fail to compute a result due to the evolution

of the internals of an Android application, raising unexpected bugs during an analysis. This chapter intends to show that sharing the software artefacts of a paper may not be sufficient to ensure that the provided software will be reusable.

Thus, our contributions are the following. We carefully retrieved static analysis tools for Android applications that were selected by Li *et al.* [33] between 2011 and 2017. We contacted the authors whenever possible to select the best candidate versions and to confirm the good usage of the tools. We rebuild the tools in their original environment and share our Docker images.[1] We evaluated the reusability of the tools by measuring the number of successful analyses of applications taken in the Drebin dataset [6] and in a custom dataset that contains more recent applications (62 525 in total). The observation of the success or failure of these analyses enables us to answer the following research questions:

RQ1 Which Android static analysis tools that are more than 5 years old are still available and can be reused without crashing with a reasonable effort?

RQ2 How has the reusability of tools evolved over time, especially when analysing applications that are more than 5 years away from the publication of the tool?

RQ3 Does the reusability of tools change when analysing goodware compared to malware?

The chapter is structured as follows. Section 3.2 presents the methodology employed to build our evaluation process, and Section 3.3 gives the associated experimental results. Section 3.4 investigates the reasons behind the observed failures of some of the tools. We then compare in Section 3.5 our results with the contributions presented in Chapter 2. In Section 3.6, we give recommendations for tool development that we drew from our experience running our experiment. Finally, Section 3.7 lists the limit of our approach, Section 3.8 presents further avenues that did not have time to pursue and Section 3.9 concludes the chapter.

3.2 Methodology

In this section, we describe our methodology to evaluate the reusability of Android static analysis tools. Figure 2 and Figure 3 summarize our approach. We collected tools listed as open source by Li *et al.*, checked if the tools were only using static analysis techniques, and selected the most recent version of the tool. We then packaged the tools inside containers and checked our choices with the authors. We then run those tools on a large dataset that we sampled, and collected the exit status of the run (whether the tool completed the analysis or not).

3.2.1 Collecting Tools

We collected the static analysis tools from [33], plus one additional paper encountered during our review of the state-of-the-art (DidFail [28]). They are listed in Table 1, with the original release date and associated publication. We intentionally limited the collected tools to the ones

1. on Docker Hub as `histausse/rasta-<toolname>:icsr2024`

Tool	Availability Bin	Src	Doc	Repo type	Decision	Comments
A3E [9] (2013)	–	✓	✓	github	✗	Hybrid tool (static/dynamic)
A5 [65] (2014)	–	✓	✗	github	✗	Hybrid tool (static/dynamic)
Adagio [21] (2013)	–	✓	✓	github	✓	
Amandroid [66] (2014)	✓	✓	✓	github	✓	
Anadroid [36] (2013)	✗	✓	✓	github	✓	
Androguard [15] (2011)	–	✓	✓✓	github	✓	
Android-app-analysis [22] (2015)	✗	✓	✓✓	google	✗	Hybrid tool (static/dynamic)
Apparecium [63] (2015)	✓	✓	✗	github	✓	
BlueSeal [58] (2014)	✗	✓	○	github	✓	
Choi *et al.* [13] (2014)	✗	✓	○	github	✗	Works on source files only
DIALDroid [12] (2017)	✓	✓	✓	github	✓	
DidFail [28] (2014)	✓	✓	○	bitbucket	✓	
DroidSafe [24] (2015)	✗	✓	✓	github	✓	
Flowdroid [8] (2014)	✓	✓	✓✓	github	✓	
Gator [54, 72] (2014, 2015)	✗	✓	✓✓	edu	✓	
IC3 [47] (2015)	✓	✓	○	github	✓	
IccTA [30] (2015)	✓	✓	✓	github	✓	
Lotrack [38] (2014)	✗	✓	✗	github	○	Authors ack. a partial doc.
MalloDroid [19] (2012)	–	✓	✓	github	✓	
PerfChecker [39] (2014)	✗	✗	○	request	✓	Binary obtained from authors
Poeplau *et al.* [51] (2014)	✗	○	✗	github	✗	Related to Android hardening
Redexer [27] (2012)	✗	✓	✓	github	✓	
SAAF [26] (2013)	✓	✓	✓	github	✓	
StaDynA [75] (2015)	✗	✓	✓	request	✗	Hybrid tool (static/dynamic)
Thresher [11] (2013)	✗	✓	✓	github	○	Not built with author's help
Wognsen *et al.* [67] (2014)	–	✓	✗	bitbucket	✓	

binaries, sources: –: not relevant, ✓: available, ○: partially available, ✗: not provided
documentation: ✓✓: excellent, MWE, ✓: few inconsistencies, ○: bad quality, ✗: not available
decision: ✓: considered; ○: considered but not built; ✗: out of scope of the study

Table 1: Considered tools [33]: availability and usage reliability

selected by Li *et al.* [33] for several reasons. First, not using recent tools enables a gap of at least 5 years between the publication and the more recent APK files, which enables measuring the reusability of previous contributions with a reasonable gap of time. Second, collecting new tools would require inspecting these tools in depth, similarly to what has been performed by Li *et al.* [33], which is not the primary goal of this chapter. Additionally, selection criteria such as the publication venue or number of citations would be necessary to select a subset of tools, which would require an additional methodology.

Some tools use hybrid analysis (both static and dynamic): A3E [9], A5 [65], Android-app-analysis [22], StaDynA [75]. They have been excluded from this study. We manually searched the

tool repository when the website mentioned in the paper is no longer available (*e.g.*, when the repository has been migrated from Google code to GitHub), and for each tool we searched for:

- an optional binary version of the tool that would be usable as a fallback (if the sources cannot be compiled for any reason).
- the source code of the tool.
- the documentation for building and using the tool with an MWE.

In Table 1 we rated the quality of these artifacts with "✓" when available but may have inconsistencies, a "○" when too much inconsistencies (inaccurate remarks about the sources, dead links or missing parts) have been found, a "✗" when no documentation have been found, and a double "✓✓" for the documentation when it covers all our expectations (building process, usage, MWE). Results show that documentation is often missing or very poor (*e.g.*, Lotrack), which makes the rebuild process very complex and the first analysis of an MWE.

We finally excluded Choi *et al.* [13] as their tool works on the sources of Android applications, and Poeplau *et al.* [51] that focus on Android hardening. As a summary, in the end, we have 20 tools to compare. Some specificities should be noted. The IC3 tool will be duplicated in our experiments because two versions are available: the original version of the authors and a fork used by other tools like IccTa. For Androguard, the default task consists of unpacking the bytecode, the resources, and the Manifest. Cross-references are also built between methods and classes. Because such a task is relatively simple to perform, we decided to duplicate this tool and ask Androguard to decompile an APK and create a control flow graph of the code using its decompiler: DAD. We refer to this variant of usage as androguard_dad. For Thresher and Lotrack, because these tools cannot be built, we excluded them from experiments.

Finally, starting with 26 tools of Table 1, with the two variations of IC3 and Androguard, we have in total 22 static analysis tools to evaluate, of which two tools cannot be built and will be considered as always failing.

3.2.2 Source Code Selection and Building Process

In a second step, we explored the best sources to be selected among the possible forks of a tool. We reported some indicators about the explored forks and our decision about the selected one in Table 2. For each source code repository called "Origin", we reported in Table 2 the number of GitHub stars attributed by users, and we mentioned if the project is still alive (✓ in column Alive when a commit exists in the last two years). Then, we analysed the fork tree of the project. We searched recursively for any forked repository that contains a more recent commit than the last one of the branch mentioned in the documentation of the original repository. If such a commit is found (the number of such commits is reported in column Alive Forks Nb), we manually looked at the reasons behind this commit and considered whether we should prefer this more up-to-date repository instead of the original one (column "Alive Forks

Tool	**Origin** Stars	Alive	**Alive Forks** Nb	Usable	**Last Commit Date**	**Authors Reached**	**Environment** Language – OS
Adagio [21]	74	✓	0	✗	2022-11-17	✓	Python – U20.04
Amandroid [66]	161	✗	2	✗	2021-11-10	✓	Scala – U22.04
Anadroid [36]	10	✗	0	✗	2014-06-18	✗	Scala/Java/Python – U22.04
Androguard [15]	4430	✓	3	✗	2023-02-01	✗	Python – Python 3.11 slim
Apparecium [63]	0	✗	1	✗	2014-11-07	✗	Python – U22.04
BlueSeal [58]	0	✗	0	✗	2018-07-04	✓	Java – U14.04
DIALDroid [12]	16	✗	1	✗	2018-04-17	✗	Java – U18.04
DidFail [28]	4	✗			2015-06-17	✓	Java/Python – U12.04
DroidSafe [24]	92	✗	3	✗	2017-04-17	✓	Java/Python – U14.04
Flowdroid [8]	868	✓	1	✗	2023-05-07	✓	Java – U22.04
Gator [54, 72]					2019-09-09	✓	Java/Python – U22.04
IC3 [47]	32	✗	3	✓	2022-12-06	✗	Java – U12.04 / 22.04
IccTA [30]	83	✗	0	✗	2016-02-21	✓	Java – U22.04
Lotrack [38]	5	✗	2	✗	2017-05-11	✓	Java – ?
MalloDroid [19]	64	✗	10	✗	2013-12-30	✗	Python – U16.04
PerfChecker [39]		✗			–	✓	Java – U14.04
Redexer [27]	153	✗	0	✗	2021-05-20	✓	Ocaml/Ruby – U22.04
SAAF [26]	35	✗	5	✗	2015-09-01	✓	Java – U14.04
Thresher [11]	31	✗	1	✗	2014-10-25	✓	Java – U14.04
Wognsen *et al.* [67]				✗	2022-06-27	✗	Python/Prolog – U22.04

✓: yes, ✗: no, UX.04: Ubuntu X.04

Table 2: Selected tools, forks, selected commits and running environment

Usable"). As reported in Table 2, we excluded all forks, except IC3, for which we selected the fork JordanSamhi/ic3, because they always contain experimental code with no guarantee of stability. For example, a fork of Aparecium contains a port for Windows 7, which does not suggest an improvement in the stability of the tool. For IC3, the fork seems promising: it has been updated to be usable on a recent operating system (Ubuntu 22.04 instead of Ubuntu 12.04 for the original version) and is used as a dependency by IccTa. We decided to keep these two versions of the tool (IC3 and IC3_fork) to compare their results.

Then, we self-allocated a maximum of four days for each tool to successfully read and follow the documentation, compile the tool and obtain the expected result when executing an analysis of an MWE. We sent an email to the authors of each tool to confirm that we used the most suitable version of the code, that the command line we used to analyse an application is the most suitable one and, in some cases, requested some help to solve issues in the building process. We reported in Table 2 the authors who answered our request and confirmed our decisions.

From this building phase, several observations can be made. Using a recent operating system, it is almost impossible in a reasonable amount of time to rebuild a tool released years ago. Too many dependencies, even for Java-based programs, trigger compilation or execution problems.

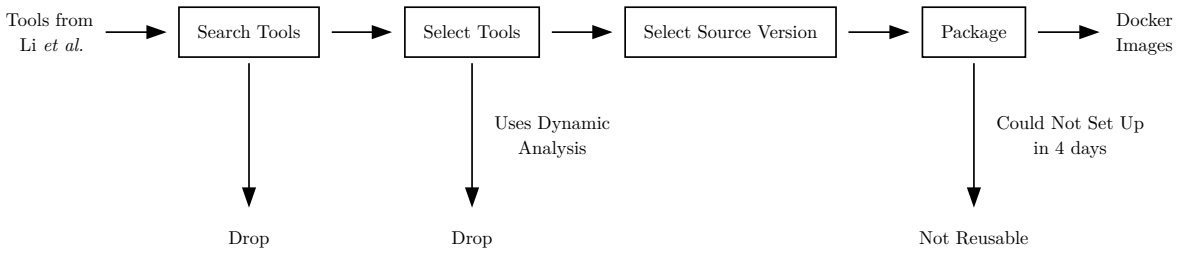


Figure 2: Tool selection methodology overview

Thus, if the documentation mentions a specific operating system, we use a Docker image of this OS.

Most of the time, tools require additional external components to be fully functional. It could be resources such as the `android.jar` file for each version of the SDK, a database, additional libraries or tools. Depending on the quality of the documentation, setting up those components can take hours to days. This is why we automated in a Dockerfile the setup of the environment in which the tool is built and run[1].

Figure 2 summarises our tool selection process. We first looked for the tools listed as open source by Li *et al.*. For the tools still available, we checked if they used dynamic analysis and removed them. We then checked if there were more recent updates of the tools and selected the most relevant version. Finally, we marked as non-reusable the tools that we could not set up within a period of 4 days, even with the help of the authors.

3.2.3 Runtime Conditions

As shown in Figure 3, before benchmarking the tools, we built and installed them in Docker containers to facilitate any reuse by other researchers. We converted them into Singularity containers because we had access to such a cluster and because this technology is often used by the HPC community for ensuring the reproducibility of experiments. We performed manual tests using these Singularity images to check:

- the location where the tool writes on the disk. For the best performances, we expect the tools to write on a mount point backed by an SSD. Some tools may write data at unexpected locations, which require small patches from us.

1. To guarantee reproducibility, we published the results, datasets, Dockerfiles and containers:
   - https://github.com/histausse/rasta .
   - https://zenodo.org/records/10144014 .
   - https://zenodo.org/records/10980349 .
   - on Docker Hub as `histausse/rasta-<toolname>:icsr2024`.

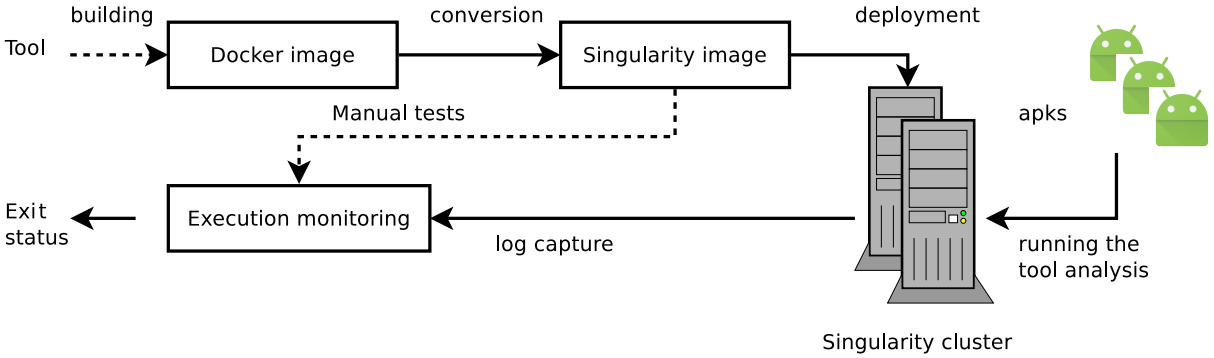


Figure 3: Experiment methodology overview

- the amount of memory allocated to the tool. We checked that the tool could run an MWE with a 64 GB limit of RAM.
- the network connection opened by the tool, if any. We expect the tool not to perform any network operation, such as the download of Android SDKs. Thus, we prepared the required files and cached them in the images during the building phase. In a few cases, we patched the tool to disable the download of resources.

A campaign of tests consists of executing the 20 selected tools on all APKs of a dataset. The constraints applied to the clusters are:

- No network connection is authorised to limit any execution of malicious software.
- The allocated RAM for a task is 64 GB.
- The allocated maximum time is 1 hour.
- The allocated object space/stack space is 64 GB / 16 GB if the tool is a Java-based program.

For the disk files, we use a mount point that is stored on an SSD disk, with no particular size limit. Note that, because the allocation of 64 GB could be insufficient for some tools, we evaluated the results of the tools on 20% of our dataset (described later in Section 3.2.4) with 128 GB of RAM and 64 GB of RAM and checked that the results were similar. With this confirmation, we continued our evaluations with 64 GB of RAM only.

3.2.4 Dataset

Two datasets are used in the experiments of this section. The first one is **Drebin** [6], from which we extracted the malware part (5479 samples that we could retrieve) for comparison purposes only. It is a well-known and very old dataset that should not be used anymore because it contains temporal and spatial biases [50]. We intend to compare the rate of success on this old dataset with a more recent one. The second one, **RASTA** (Reusability of Android Static Tools and Analysis), we built to cover all dates between 2010 and 2023. This dataset is a random extract of Androzoo [3], for which we balanced applications between years and size. For each

year and inter-decile range of size in Androzoo, 500 applications have been extracted with an arbitrary proportion of 7% of malware. This ratio has been chosen because it is the ratio of goodware/malware that we observed when performing a raw extract of Androzoo. For checking the maliciousness of an Android application, we rely on the VirusTotal detection indicators. If more than 5 antiviruses have flagged the application as malicious, we consider it malware. If no antivirus has reported the application as malicious, we consider it goodware. Applications in between are dropped.

For computing the release date of an application, we contacted the authors of Androzoo to compute the minimum date between the submission to Androzoo and the first upload to VirusTotal. Such a computation is more reliable than using the DEX date that is often obfuscated when packaging the application.

3.3 Experiments

3.3.1 RQ1: Re-Usability Evaluation

Figure 4 and Figure 5 compare the Drebin and RASTA datasets. They represent the success/failure rate (green/orange) of the tools. We distinguished failure to compute a result from timeout (blue) and crashes of our evaluation framework (in grey, probably due to out-of-memory kills of the container itself). Because it may be caused by a bug in our own analysis stack, exit statuses represented in grey (Other) are considered as unknown errors and not as failures of the tool. We discuss further errors for which we have information in the logs in Section 3.4.

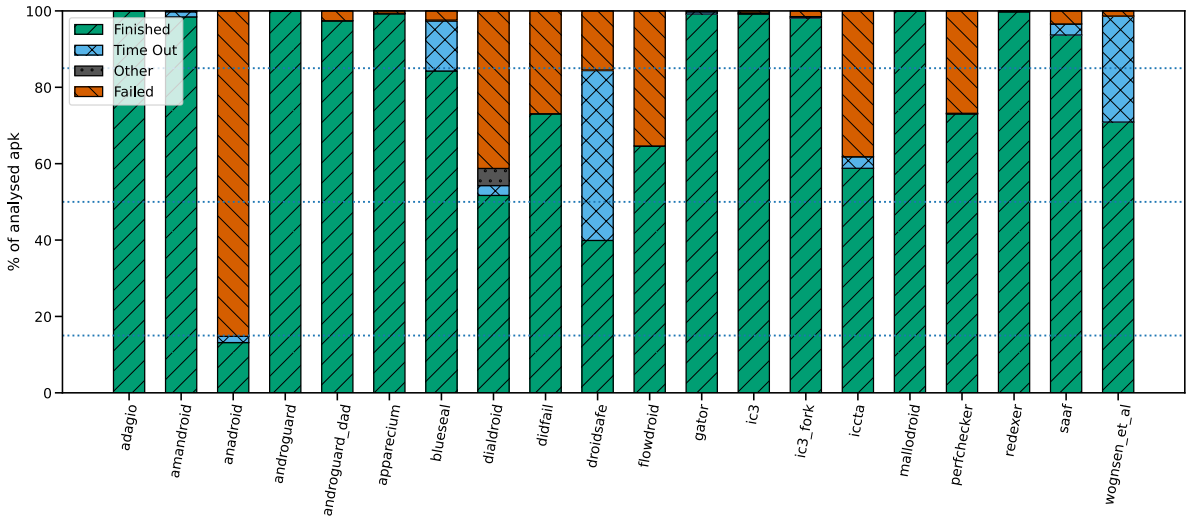


Figure 4: Exit status for the Drebin dataset

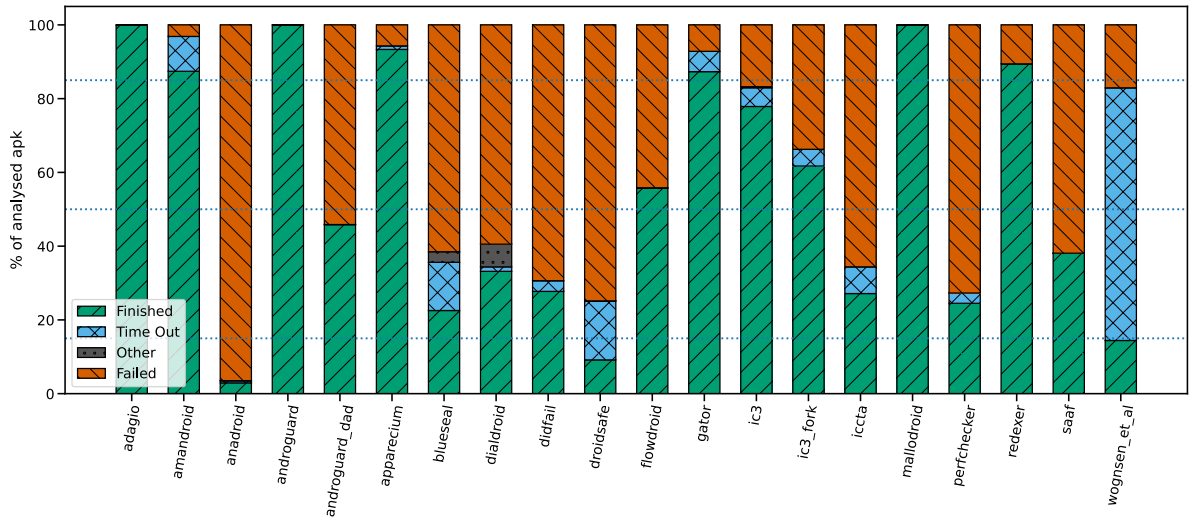


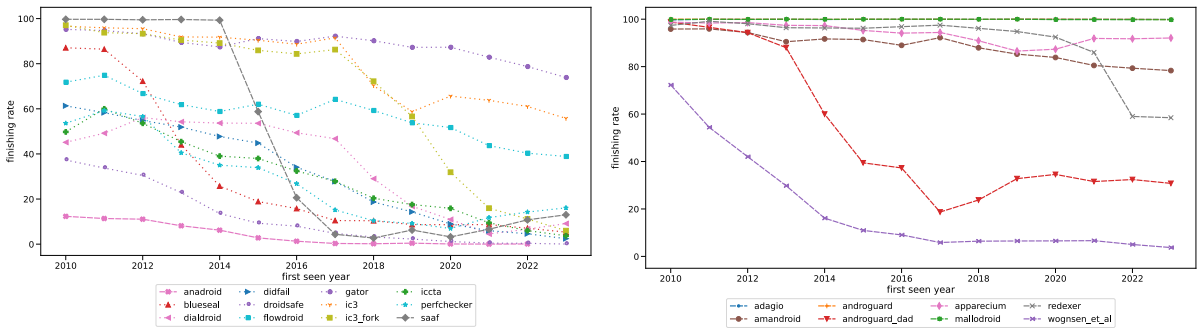
Figure 5: Exit status for the RASTA dataset

Results on the Drebin datasets show that 11 tools have a high success rate (greater than 85%). The other tools have poor results. The worst, excluding Lotrack and Tresher, is Anadroid with a ratio under 20% of success.

On the RASTA dataset, we observe a global increase in the number of failed status: 12 tools (54.55%) have a finishing rate below 50%. The tools that have bad results with Drebin are, of course, bad results on RASTA. Three tools (androguard_dad, blueseal, saaf) that were performing well (higher than 85%) on Drebin, surprisingly fall below the bar of 50% of failure. 7 tools keep a high success rate: Adagio, Amandroid, Androguard, Apparecium, Gator, Mallodroid, Redexer. Regarding IC3, the fork with a simpler build process and support for modern OS has a lower success rate than the original tool.

Two tools should be discussed in particular. Androguard has a high success rate, which is not surprising: it is used by a lot of tools, including for analysing applications uploaded to the Androzoo repository. Nevertheless, when using Androguard decompiler (DAD) to decompile an APK, it fails more than 50% of the time. This example shows that even a tool that is frequently used can still run into critical failures. Concerning Flowdroid, our results show a very low timeout rate (0.06%), which was unexpected: in our exchanges, Flowdroid's authors were expecting a higher rate of timeout and fewer crashes.

As a summary, the final ratio of successful analysis for the tools that we could run is 54.9%. When including the two defective tools, this ratio drops to 49.9%.



a) Java-based tools

b) Non-Java-based tools

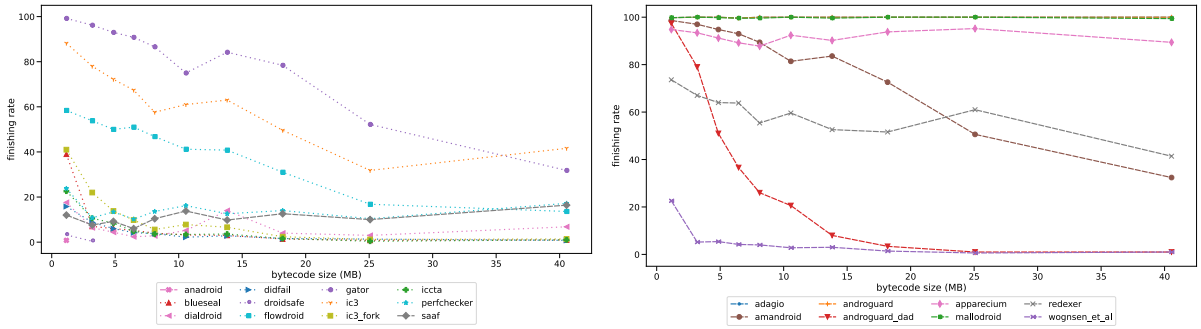
Figure 6: Exit status evolution for the RASTA dataset

RQ1 answer: On a recent dataset, we consider that 54.55% of the tools are unusable. For the tools that we could run, 54.9% of analyses are finishing successfully.

3.3.2 RQ2: Size, SDK and Date Influence

For investigating the effect of application dates on the tools, we computed the date of each APK based on the minimum date between the first upload in AndroZoo and the first analysis in VirusTotal. Such a computation is more reliable than using the DEX date, which is often obfuscated when packaging the application. Then, for the sake of clarity of our results, we separated the tools that have mainly Java source code from those that use other languages. Among the ones that are Java-based programs, most of them use the Soot framework, which may correlate the obtained results. Figure 6 a) (resp. Figure 6 b)) compares the success rate of the tools between 2010 and 2023 for Java-based tools (resp. non Java-based tools). For Java-based tools, a clear decrease in finishing rate can be observed globally for all tools. For non-Java-based tools, 2 of them keep a high success rate (Androguard, Mallodroid). The result is expected for Androguard, because the analysis is relatively simple and the tool is largely adopted, as previously mentioned. Mallodroid, being a relatively simple script leveraging Androguard, benefits from Androguard's resilience. It should be noted that Saaf kept a high success ratio until 2014 and then quickly decreased to less than 20% after 2014. This example shows that, even with an identical source code and the same running platform, a tool can change its behaviour over time because of the evolution of the structure of the input files.

An interesting comparison is the specific case of IC3 and Ic3_fork. Until 2019, the success rate was very similar. After 2020, ic3_fork is continuing to decrease, whereas IC3 keeps a success rate of around 60%.



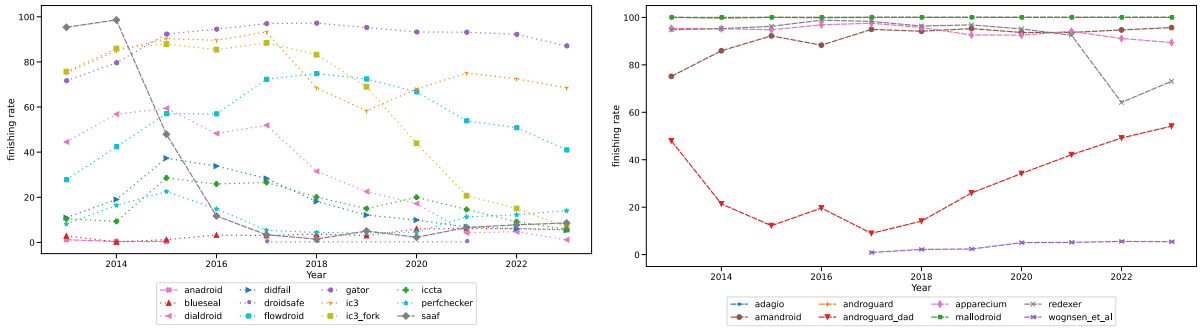
a) Java-based tools b) Non-Java-based tools

Figure 7: Finishing rate by bytecode size for APK detected in 2022

To compare the influence of the date, SDK version and size of applications, we fixed one parameter while varying another.

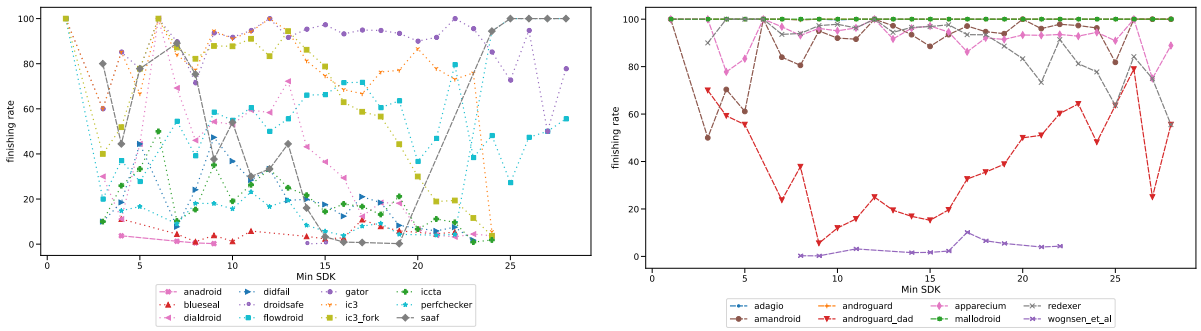
Fixed application year. (5000 APKs) We selected the year 2022, which has a good amount of representatives for each decile of size in our application dataset. Figure 7 a) (resp. Figure 7 b)) shows the finishing rate of the tools in function of the size of the bytecode for Java-based tools (resp. non-Java-based tools) analysing applications of 2022. We can observe that all Java-based tools have a finishing rate that decreases over the years. 50% of non-Java-based tools have the same behaviour.

Fixed application bytecode size. (6252 APKs) We selected the sixth decile (between 4.08 and 5.20 MB), which is well represented in a wide number of years. Figure 8 a) (resp. Figure 8 b)) represents the finishing rate depending on the year at a fixed bytecode size. We observe that 9



a) Java-based tools b) Non-Java-based tools

Figure 8: Finishing rate by discovery year with a bytecode size ∈ [4.08, 5.2] MB



a) Java-based tools

b) Non-Java-based tools

Figure 9: Finishing rate by min SDK with a bytecode size ∈ [4.08, 5.2] MB

tools out of 12 have a finishing rate dropping below 20% for Java-based tools, which is not the case for non-Java-based tools.

We performed similar experiments by varying the min SDK and target SDK versions, still with a fixed bytecode size between 4.08 and 5.2 MB, as shown in Figure 9 a) and Figure 9 b). We found that, contrary to the target SDK, the min SDK version has an impact on the finishing rate of Java-based tools: 8 tools over 12 are below 50% after SDK 16. It is not surprising, as the min SDK is highly correlated to the year.

RQ2 answer: For the 20 tools that can be used partially, a global decrease in the success rate of tools' analysis is observed over time. Starting at a 78% success rate, after five years, tools have 61% success; after ten years, 45% success. The success rate varies based on the size of the bytecode and SDK version. The date is also correlated with the success rate for Java-based tools only.

3.3.3 RQ3: Malware vs Goodware

RASTA part	**Average size** (MB) **APK**	**DEX**
goodware	16.9	6.6
malware	17.2	4.3
total	16.9	6.5

Table 3: Average size and date of goodware/malware parts of the RASTA dataset

We sampled our dataset to have a variety of APK sizes, but the size of the application is not entirely proportional to the bytecode size. Looking at Table 3, we can see that although malware are, on average, bigger APKs, they contain less bytecode than goodware. In the previous section,

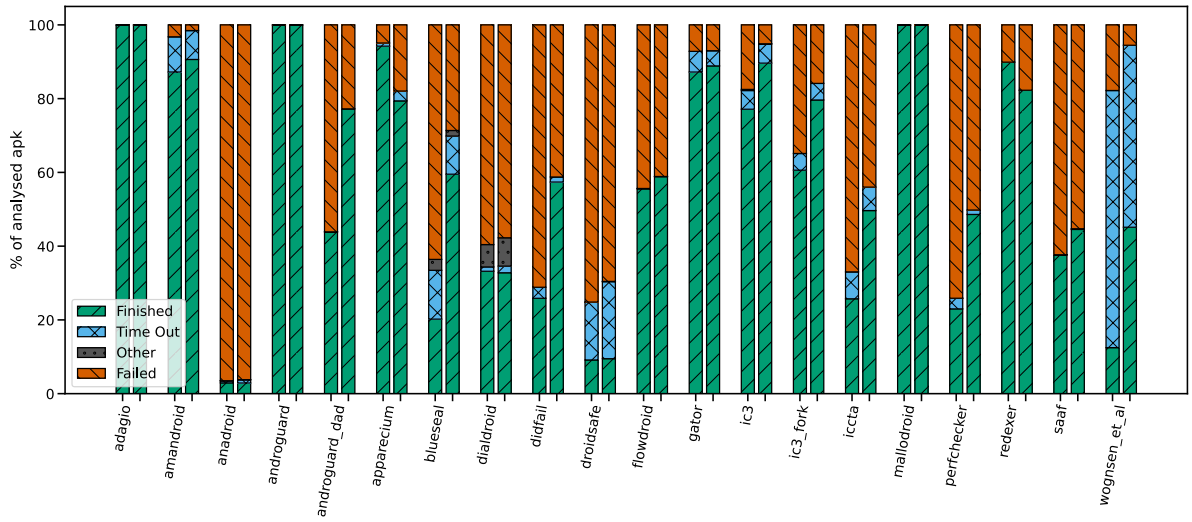


Figure 10: Exit status comparing goodware (left bars) and malware (right bars) for the RASTA dataset

we saw that the size of the bytecode has the most significant impact on the finishing rate of analysis tools, and indeed, Figure 10 reflects that.

In Figure 10, we compared the finishing rate of malware and goodware applications for the evaluated tools. We can see that malware and goodware seem to generate a similar number of timeouts. However, with the exception of two tools – apparecium and redexer, we can see a trend of goodware being harder to analyse than malware. Some tools, like DAD or perfchecker, show the finishing rate ratio augmented by more than 20 points.

Decile	**Average DEX size (MB)** Good	Mal	**Finishing Rate: FR** Good	Mal	**Ratio Size** Good/Mal	**Ratio FR** Good/Mal
1	0.13	0.11	0.85	0.82	1.17	1.04
2	0.54	0.55	0.74	0.72	0.97	1.03
3	1.37	1.25	0.63	0.66	1.09	0.97
4	2.41	2.34	0.57	0.62	1.03	0.92
5	3.56	3.55	0.53	0.59	1	0.9
6	4.61	4.56	0.5	0.61	1.01	0.82
7	5.87	5.91	0.47	0.57	0.99	0.83
8	7.64	7.63	0.43	0.56	1	0.76
9	11.39	11.26	0.39	0.58	1.01	0.67
10	24.24	21.36	0.33	0.46	1.13	0.73

Table 4: DEX size and Finishing Rate (FR) per decile

We saw that the bytecode size may be an explanation for this increase. To investigate this further, Table 4 reports the bytecode size and the finishing rate of goodware and malware in each decile of bytecode size. We also computed the ratio of the bytecode size and finishing rate for the two populations. We observe that while the bytecode size ratio between goodware and malware stays close to one in each decile (excluding the two extremes), the goodware/malware finishing rate ratio decreases for each decile. It goes from 1.03 for the 2nd decile to 0.67 in the 9th decile. We conclude from this table that, at equal size, analysing malware still triggers fewer errors than for goodware, and that the difference in errors generated between when analysing goodware and analysing malware increases with the bytecode size.

RQ3 answer: Analysing malware applications triggers fewer errors for static analysis tools than analysing goodware for comparable bytecode size.

3.4 Failures Analysis

In this section, we investigate the reasons behind the high failure ratio presented in Section 3.3. Table 5 reports the average number of errors, the average time and memory consumption of the analysis of one APK file.

3.4.1 Detected Errors

During the running of our experiment, we parsed the standard output and error to capture:

- Java errors and stack traces.
- Python errors and stack traces.
- Ruby errors and stack traces.
- Log4j messages with a “ERROR” or “FATAL” level.
- XSB error messages.
- Ocaml errors.

For example, Dialdroid reports an average of 55.9 errors for one successful analysis. On the contrary, some tools, such as Blueseal report very few errors at a time, making it easier to identify the cause of the failure.

Because some tools send back a high number of errors in our logs (up to 46 698 for one execution), we tried to determine the error that is linked to the failed status. Unfortunately, our manual investigations confirmed that the last error of a log output is not always the one that should be attributed to the global failure of the analysis. The error that seems to generate the failure can occur in the middle of the execution, be caught by the code and then other subsequent parts of the code may generate new errors as consequences of the first one. Similarly, the first error in the logs is not always the cause of a failure. Sometimes errors successfully caught and handled are logged anyway. Thus, it is impossible to accurately extract the error responsible for

Exit status		adagio	amandroid	anadroid	androguard	androguard_dad	apparecium	blueseal	dialdroid	didfail	droidsafe	flowdroid	gator	ic3	ic3_fork	iccta	mallodroid	perfchecker	redexer	saaf	wognsen_et_al
Average number of errors (and standard deviation)																					
FINISHED	errors	0	0.9	0.02	0	0	3.33	0	55.88	1.13	7.17	1.04	0	1.22	0.32	14.3	4.37	0.28	1.29	0.22	2.13
	σ	0	3.23	0.19	0	0	261.92	0.05	63.73	3.94	37.87	26.32	0.04	23.1	2.73	71.74	277	1.77	1.28	0.83	66.5
FAILED	errors	0	2.79	2.34	1.35	1	21.63	1.02	33.79	6.6	12.53	14.64	0.32	3.66	1.29	17.34	1	1.15	3.45	6.35	4.11
	σ	0	8.7	0.94	0.48	0.02	466.97	0.21	108.56	31.56	74.01	49.07	0.78	18.06	0.71	42.81	0	4.7	4.52	22.97	48.81
TIMEOUT	errors	0	9.78	0.01	0	0	4.3	0.01	60.94	1.06	26.64	0.75	0	2.13	0.91	3.68	0	1.24	0	91.29	1.31
	σ	0	9.76	0.11	0	0	79.98	0.11	101.73	2.98	97.18	1.72	0	5.19	3.19	15.33	0	4.3	0	353.75	3.42
Average time (s)																					
FINISHED		17	405	149	16	27	98	158	768	270	676	29	33	156	159	90	28	4	16	56	696
FAILED	**time**	8	761	5	14	63	22	12	68	445	443	137	924	535	29	202	5	10	17	6	56
TIMEOUT		0	3601	3601	0	3604	3600	3601	3604	3600	3600	3601	3601	3601	3601	3600	0	3600	0	3602	3601
Average Memory (GB)																					
FINISHED		0.6	4.5	12.8	0.6	0.3	1.3	2.7	15.9	17.6	9.9	2	2	15.3	5	5	0	0	1	3	3
FAILED	**memory**	0.3	4.9	2.8	0.4	1	0.6	1.7	3.9	68.3	14.8	5	41.5	130.9	5	12	0	1	1	1	1
TIMEOUT		0	19	82.6	0	68.1	2.1	15.4	37.2	99.8	0.2	20.2	1.1	81.1	20	2	0	1	0	9	0

Table 5: Average number of errors, analysis time, memory per unitary analysis – compared by exit status

a failed execution. Therefore, we investigated the nature of errors globally, without distinction between error messages in a log.

Figure 11 draws the most frequent error objects for each of the tools. A black square is an error type that represents more than 80% of the errors raised by the considered tool. In between, grey squares show a ratio between 20% and 80% of the reported errors.

First, the heatmap helps us to confirm that our experiment is running in adequate conditions. Regarding errors linked to memory, two errors should be investigated: `OutOfMemoryError` and `StackOverflowError`. The first one only appears for Gator with a low ratio. Several tools have a low ratio of errors concerning the stack. These results confirm that the allocated heap and stack are sufficient for running the tools with the RASTA dataset. Regarding errors linked to the disk space, we observe small ratios for the exception `IOException`, `FileNotFoundError` and `FileNotFoundException`. Manual inspections revealed that those errors are often a consequence of a failed Apktool execution.

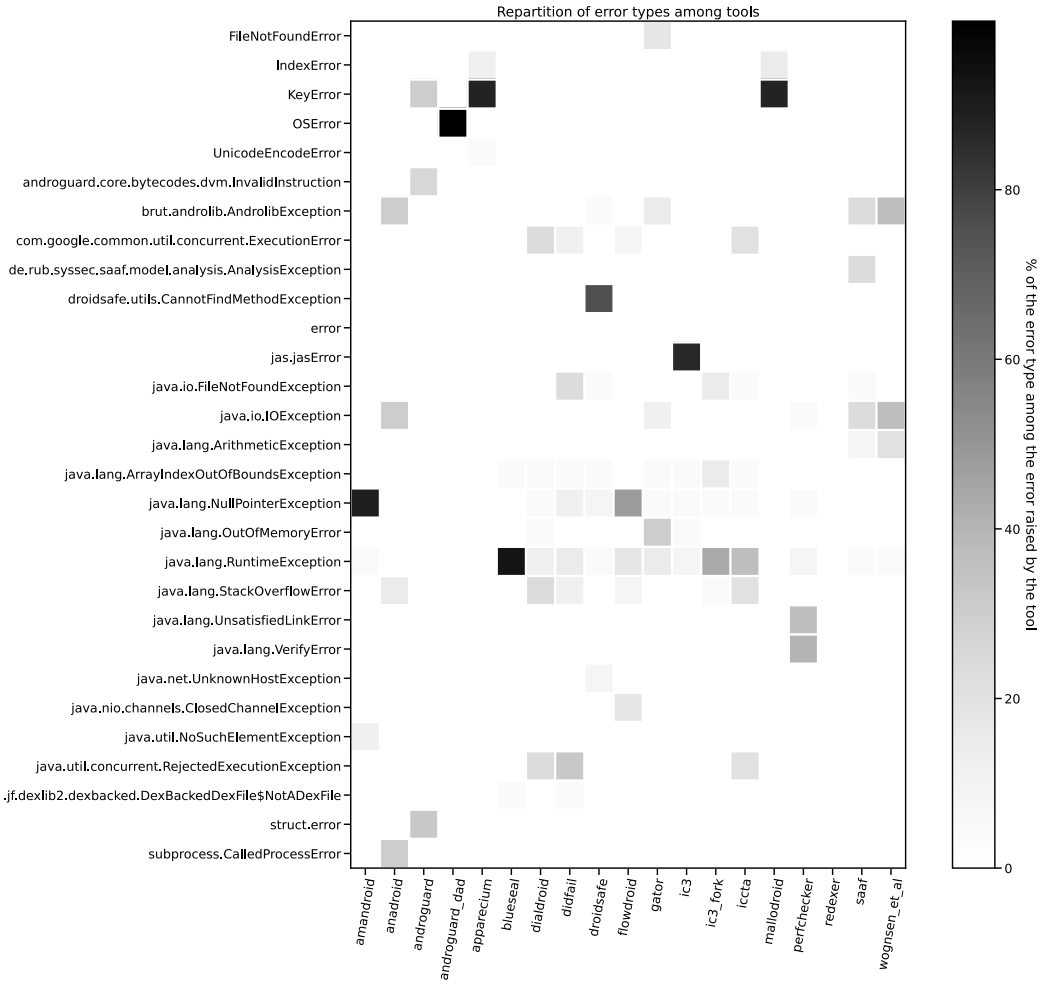


Figure 11: Heatmap of the ratio of error reasons for all tools for the RASTA dataset

Second, the black squares indicate frequent errors that need to be investigated separately. In the next subsection, we manually analysed, when possible, the code that generates these high ratios of errors, and we give feedback about the possible causes and difficulties in writing a bug fix.

3.4.2 Tool by Tool Investigation

Androguard and Androguard_dad Surprisingly, while Androguard rarely fails to analyse an APK, the internal decompiler of Androguard (DAD) fails more than half of the time. The

analysis of the logs shows that the issue comes from the way the decompiled methods are stored: each method is stored in a file named after the method name and signature, and this file name can quickly exceed the size limit (255 characters on most file systems). It should be noticed that Androguard_dad rarely fails on the Drebin dataset. This illustrates the importance of testing tools on real and up-to-date APKs: even a bad handling of filenames can influence an analysis.

Mallodroid and Apparecium Mallodroid and Apparecium stand out as the tools that raised the most errors in one run. They can raise more than 10 000 errors by analysis. However, it happened only for a few dozen APKs, and conspicuously, the same APKs raised the same high number of errors for both tools. The recurring error is a `KeyError` raised by Androguard when trying to find a string by its identifier. Although this error is logged, it seems successfully handled, and during a manual analysis of the execution, both tools seemingly perform their analysis without issue. This high number of occurrences may suggest that the output is not valid. Still, the tools claim to return a result, so, from our perspective, we consider those analyses as successful. For numerous other errors, we could not identify the reason why those specific applications raise so many exceptions. However, we noticed that Mallodroid and Apparecium use an outdated version of Androguard (respectively version 3.0 and 2.0), and neither Androguard v3.3.5 nor DAD with Androguard v3.3.5 raise those exceptions. This suggests the issue has been fixed by Androguard and that Mallodroid and Apparecium could benefit from a dependency upgrade.

Blueseal Because Blueseal rarely logs more than one error when crashing, it is easy to identify the relevant error. The majority of crashes come from unsupported Android versions (due to the magic number of the DEX files not being supported by the version of back smali used by Blueseal) and methods whose implementation is not found (like native methods).

Droidsafe and SAAF Our investigation of the most common errors raised by Droidsafe and SAAF showed that they are often preceded by an error from apktool. Indeed, 28 654 runs of Droidsafe and 38 635 runs of SAAF failed after raising at least one of `brut.androlib.AndrolibException` or `brut.androlib.err.UndefinedResObject`, suggesting that those tools would benefit from an upgrade of apktool.

IC3 and IC3_fork We compared the number of errors between IC3 and IC3_fork. IC3_fork reports fewer errors for all types of analysis, which suggests that the author of the fork has removed the outputted errors from the original code: the thrown errors are captured in a generic `RuntimeException`, which removes the semantics, making it harder for our investigations. Nevertheless, IC3_fork has more failures than IC3: the number of errors reported by a tool is not correlated to the final success of its analysis.

Flowdroid Our exchanges with the authors of Flowdroid led us to expect more timeouts from executions taking too long than failed runs. Surprisingly, we only got 0.06% of timeout, and a high number of failures. We tried to detect recurring causes of failures, but the complexity

of Flowdroid makes the investigation difficult. Most exceptions seem to be related to concurrency. Other errors that came up regularly are `java.nio.channels.ClosedChannelException`, which is raised when Flowdoid fails to read from the APK, although we did not find the reason for failure, null pointer exceptions when trying to check if a null value is in a `ConcurrentHashMap` (in `LazySummaryProvider.getClassFlows()`) and `StackOverflowError` from `StronglyConnectedComponentsFast.recurse()`. We randomly selected 20 APKs that generated stack overflows in Flowdroid and retried the analysis with 500GB of RAM allocated to the JVM. 18 of those runs still failed with a stack overflow without using all the allocated memory, and the other two failed after raising null pointer exceptions from `getClassFlows`. This shows that the lack of memory is not the primary cause of those failures.

As a conclusion, we observe that a lot of errors can be linked to bugs in dependencies. Our attempts to upgrade those dependencies led to new errors appearing: we conclude that this is not a trivial task that requires familiarity with the inner code of the tools.

3.5 State-of-the-Art Comparison

In this section, we will compare our results with the contributions presented in Chapter 2.

Luo *et al.* released TaintBench [40], a real-world benchmark and the associated recommendations to build such a benchmark. These benchmarks confirmed that some tools, such as Amandroid and Flowdroid, are less efficient on real-world applications. We confirm the hypothesis of Luo *et al.* that real-world applications lead to less efficient analysis than using handcrafted test applications or old datasets [40]. In addition, even if Drebin is not hand-crafted, it is quite old and seems to present similar issues as handcrafted datasets when used to evaluate a tool: we obtained really good results compared to the RASTA dataset – which is more representative of real-world applications.

Our findings are also consistent with the numerical results of Pauck *et al.* that showed that 58.89% of DIALDroid-Bench [12] real-world applications are analysed successfully with the 6 evaluated tools [49]. Six years after the release of DIALDroid-Bench, we obtain a lower ratio of 40.05% for the same set of 6 tools but using the RASTA dataset of 62 525 applications. We extended this result to a set of 20 tools and obtained a global success rate of 54.9%. We confirmed that most tools require a significant amount of work to get them running [53]. Our investigations of crashes also confirmed that dependencies on older versions of Apktool are impacting the performances of Anadroid, Saaf and Wognsen *et al.* in addition to DroidSafe and IccTa, already identified by Pauck *et al.*.

Third, we extended to 20 different tools the work done by Reaves *et al.* on the usability of analysis tools (4 tools are in common, we added 16 new tools and two variations). We confirmed that most tools require a significant amount of work to get them running. We encounter similar issues with libraries and operating system incompatibilities, and noticed that, as time passes,

dependency issues may impact the build process. For instance, we encountered cases where the repositories hosting the dependencies were closed, or cases where Maven failed to download dependencies because the OS version did not support SSL, which is now mandatory to access Maven Central.

3.6 Recommendations

In light of our findings in Section 3.4 and the issues we encountered while packaging the tools, we summarise some takeaways that we believe developers should follow to improve the success of reusing their software.

We understand software developed for research purposes is not and should not be held to the same standards as production software. However, research is incremental, and it is not sustainable to start each tool from scratch. It is critical to be able to build upon tools already published, and efforts should be made to allow that when releasing a tool.

During the packaging and testing of the tools we examined in our experiment, the most notable issues we encountered could have been avoided by following classical development best practices. To make a tool easy to reuse, it should have documentation with at least:

- Instructions about how to install the dependencies.
- Instructions about how to build the tool (if the tool needs to be built).
- Instructions about how to use the tool (*e.g.*, command line arguments).
- Instructions about how to interpret the results of the tools (we only checked for the existence of the results in our experiment, but we found that some results can be quite obscure).

In addition to the documentation, a minimum working example with the expected result of the tools allows a potential user to check if everything is working as intended. This MWE have the additional benefit that it can serve as an example in the documentation.

Another best practice to follow is to pin the version of dependencies of the tools. Many modern dependency management tools can handle that: for instance, for Python, Poetry or uv generate a lock file with the exact version of the libraries to use, Cargo does the same for Rust, in Java, this can be an option in Gradle, and dependencies in Maven `pom.xml` files are usually the exact version. For other dependencies that are not managed by a dependency manager – for instance, the Java virtual machine to use, the Python interpreter, resource files – the version to use should be clearly documented. Alternatively, tools like nixpkg can be used to pin every dependency. The worst case we encountered during our experiment was a tool whose documentation instructed us to install the z3 dependencies with a simple `git clone`, without specifying the commit to use. The z3 project is still actively maintained, so the dependency installed was not compatible, and finding a compatible version required checking releases one by one. Dependencies fetched with a version control system should always indicate the exact version to use (in the case of git, a commit, tag, or release should be used).

We also found that interactions with the running environment can become very problematic when the environment changes. To minimise the issues, packaging the tool inside a Docker container or even a virtual machine can ensure that future users have at least access to a working version of the tool.

Finally, when possible, continuous integration, tests and code reviews should be implemented to improve the reliability of the developed tool.

Concerning the actual code of the tool, more attention should be paid to error reporting. When a tool failed to perform its analysis, it should be clear to the user, and the reason should be clearly reported. In some cases, this may imply *not* trying to recover from unrecoverable errors: this often leads to errors seemingly unrelated to the initial issue. This is often a problem in Java code, where the developers are strongly encouraged to catch all exceptions, and in bash scripts that run several programs in a row without checking the exit statuses.

Good error reporting can allow future users to solve issues encountered using the tools: for instance, the log generated by Androguard's decompiler clearly shows that the issue is file names exceeding the size limit. This issue could easily be fixed by changing the filenames used to store the results. In contrast, the errors generated by Flowdroid are so opaque that we have no idea how we could solve them.

At last, an important remark concerns the libraries used by a tool. We have seen two types of libraries:

- internal libraries manipulating internal data of the tool.
- external libraries that are used to manipulate the input data (APKs, bytecode, resources).

We observed during our manual investigations that external libraries are the ones leading to crashes because of variations in recent APKs (file format, unknown bytecode instructions, multi-DEX files). We believe that the developer should provide enough documentation to make a later upgrade of these external libraries possible. For example, old versions of Apktool are the top-most libraries raising errors, but breaking changes introduced by upgrading from v1.X versions to v2.X versions prevent us from upgrading Apktool.

3.7 Limitations

Some limitations of our approach should be kept in mind.

Our application dataset is biased in favour of Androguard, because Androzoo has already used Androguard internally when collecting applications and discarded any application that cannot be processed with this tool.

Despite our best efforts, it is possible that we made mistakes when building or using the tools. It is also possible that we wrongly classified a result as a failure. To mitigate this possible

problem, we contacted the authors of the tools to confirm that we used the right parameters and chose a valid failure criterion. Before running the final experiment, we also ran the tools on a subset of our dataset and manually investigated the most common errors to ensure that they are not trivial errors that can be solved.

The timeout value and memory limits are arbitrarily fixed. To mitigate this issue, a small extract of our dataset has been analysed with more memory/time, and we checked that there was no significant difference in the results.

Finally, the use of VirusTotal for determining if an application is malware or not may be wrong. To limit the impact of errors, we used a threshold of at most 5 antiviruses (resp. no more than 0) reporting an application as being malware (resp. goodware) for taking a decision about maliciousness (resp. benignness).

3.8 Future Works

A first extension to this work would obviously be to study more tools. We restricted ourselves to the tools listed by Li *et al.*, but it would be interesting to compare our result to the finishing rate of recently released tools. It would be interesting to see if they are better at handling large APKs, but also to see if older applications are more challenging for them due to discontinued features.

Another avenue would be to define a benchmark to check the ability of tools to handle real-world applications. Our dataset is too large for a simple benchmark and is sampled to have a variety of application sizes and years of publication. Hence, the first step would be to sample a dataset for this benchmark. Current benchmark datasets focus on the accuracy of the tested tools, with difficult-to-analyse applications. It could be interesting to extract from our results some of the applications that the most tools failed to analyse, and either use them directly or study them to craft simpler applications reproducing the same challenges as those applications. Such datasets would need to be updated regularly: we saw that there is a trend for newer applications to be harder to analyse, a frozen dataset would ignore this factor.

In addition to the finishing rate, it would be both interesting and useful to have reference values. Table 6 list common Android-related dependencies we encountered when packaging the tools. We can see that each tool uses at least one of those dependencies. It would be reasonable to consider the best finishing ratio a tool can have to be the finishing ratio of a tool that would perform an “empty analysis” using the same dependencies. Considering the prevalence of those dependencies, having those theoretical minimums could also guide future tool developers when choosing their dependencies.

Tool	Soot	Androguard	Apktool
adagio		✓	
amandroid			✓
anadroid			✓
androguard		✓	
androguard_dad		✓	
apparecium		✓	
blueseal	✓		✓
dialdroid	✓		
didfail	✓		
droidsafe	✓		✓
flowdroid	✓		
gator	✓		✓
ic3	✓		
ic3_fork	✓		
iccta	✓		✓
mallodroid		✓	
perfchecker	✓		
redexer			✓
saaf			✓
wognsen_et_al			✓

Table 6: Commonly found dependencies

3.9 Conclusion

Since the release of Android, many tools have been published in order to analyse Android applications. In Chapter 2, we went through contributions that benchmark and compare some of those tools. Those contributions suggested that analysing real-world applications might be more challenging than expected. This led us to question the reusability of those tools (**Pb1**).

This chapter has assessed the suggested results of the literature [40, 49, 53] about the reliability of static analysis tools for Android applications. With a dataset of 62 525 applications, we established that 54.55% of 22 tools are not reusable. 2 of those were due to the fact that we did not manage to use the tools, even with the help of the author. We consider the 10 other tools to be unusable due to the fact that they fail to finish their analysis more than 50% of the time.. In total, the analysis success rate of the tools that we could run for the entire dataset is 54.9%. The characteristics that have the most influence on the success rate are the bytecode size and the min SDK version. Finally, we showed that malware APKs generate fewer fatal errors than goodware when analysed.

Following Reaves *et al.* recommendations [53], we publish the Docker and Singularity images we built to run our experiments alongside the Docker files. This will allow the research community to use the tools directly without the build and installation penalty.

Pb1: *To what extent are previously published Android analysis tools still usable today, and what factors impact their reusability?*

More than half the tools we selected were not usable. In some cases, it was due to our inability to set up the tool correctly. Mostly, it was due to the high failure rate when analysing real-world applications. Results show that large applications cause more crashes, as do applications with a higher min SDK target. Goodware also appears to generate more analysis failures than malware.

Chapter 4

Class Loaders in the Middle: Confusing Android Static Analysers

---

Things that try to look like things often do look more like things than things.

— Esmerelda Weatherwax, Wyrd Sisters, Terry Pratchett

The dynamic linking and loading of the different classes by the ART is a complex task that can eventually be exploited by an attacker. In particular, if the developer adds a class whose name collides with the name of a class of the Android operating system or another class in the application, they may confuse a reverse engineer in charge of studying such an application. In this chapter, we explore the consequences of those collisions. We highlight three attacks that we call shadow attacks because the class implementation that a reverser would find shadows a second implementation with a higher priority. In particular, we show that static analysis tools used by a reverser choose the shadow implementation for most of the evaluated tools, and output a wrong result. In a dataset of 49 975 applications, we also investigate whether shadow attacks are used in the wild and show that, most of the time, there is no malicious behaviour behind them.

4.1 Introduction

In this chapter, we study how Android handles the loading of classes in the case of multiple versions of the same class. Such collisions can exist inside the APK file or between the APK

file and Android SDK classes. We intend to understand if a reverser would be impacted during a static analysis when dealing with such obfuscated code. Because this problem is already complex enough with the current operations performed by Android, we exclude the case where the application uses class loaders dynamically (*e.g.*, dynamic code loading). We present a new technique that "shadows" a class, *i.e.*, embeds a class in the APK file and "presents" it to the reverser instead of the legitimate version. We show how these attacks can confuse the tools of the reverser when he performs a static analysis, and, in order to evaluate if such attacks are already used in the wild, we analysed 49 975 applications from 2023 that we extracted randomly from AndroZoo [3].

The chapter is structured as follows. Section 4.2 investigates the internal mechanisms of class loading and presents how a reverser can be confused by these mechanisms. Then, in Section 4.3, we design obfuscation techniques and show their effect on static analysis tools. Next, Section 4.4 evaluates whether these obfuscation techniques are used in the wild by scanning 49 975 APKs. Section 4.6 extends on the possible countermeasures against those shadow attacks, how they interact with other obfuscation techniques, as well as the limitations of this work and avenues left to explore. Finally, Section 4.6 concludes the chapter.

4.2 Analysing the Class Loading Process

For building obfuscation techniques based on the confusion of tools with class loaders, we manually studied the code of Android that handles class loading. In this section, we report the inner workings of ART, and we focus on the specificities of class loading that can bring confusion. Because the class loading implementation has evolved over time during the multiple iterations of the Android operating system, we mainly describe the behaviour of ART from Android version 14 (SDK 34).

4.2.1 Class Loaders

When ART needs to access a class, it queries an object implementing the `ClassLoader` class to retrieve its implementation. Each class has a reference to the `ClassLoader` that loaded it, and this class loader is the one that will be used to load supplementary classes used by the original class. For example, in Listing 1, when calling `A.f()`, the ART will load `B` with the class loader that was used to load `A`.

```
1 class A {
2     public static void f() {
3         B b = new B();
4         b.do_something();
5 }}
```

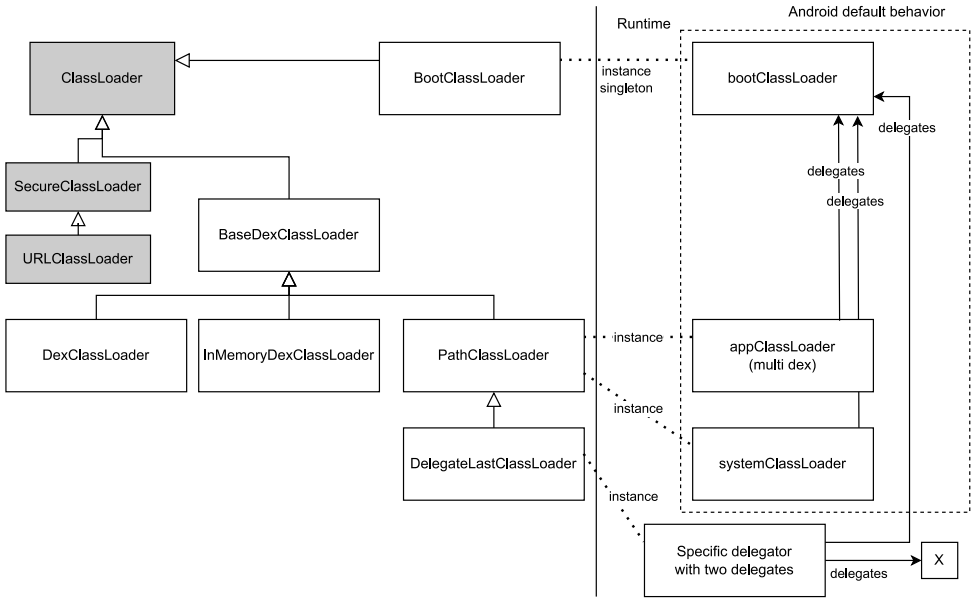
Listing 1: Class instantiation

This behaviour has been inherited from Java, and most of the core classes regarding class loaders have been kept in Android. Nevertheless, the Android implementation has slight differences, and new class loaders have been added. For example, the Java class loader `URLClassLoader` is still present in Android, but contrary to the official documentation, most of its methods have been removed or replaced by a stub that just raises an exception. Moreover, rather than using the Java class loaders `SecureClassLoader` or `URLClassLoader`, Android has several new class loaders that inherit from `ClassLoader` and override the appropriate methods.

The left part of Figure 12 shows the different class loaders specific to Android in white and the stubs of the original Java class loaders in grey. The main difference between the original Java class loaders and the ones used by Android is that they do not support the Java bytecode format. Instead, the Android-specific class loaders load their classes from (many) different file formats specific to Android. Usually, when used by a programmer, the classes are loaded from memory or from a file using the DEX format (`.dex`). When used directly by ART, the classes are usually stored in an application file (`.apk`) or in an optimised format (`OAR/ODEX`).

4.2.2 Delegation

The order in which classes are loaded at runtime requires special attention. All the specific Android class loaders (`DexClassLoader`, `InMemoryClassLoader`, etc.) have the same behaviour (except `DelegateLastClassLoader`), but they handle specificities for the input format. Each



grey – Java-based, white – Android-based
Figure 12: The class loading hierarchy of Android

class loader has a delegate class loader, represented in the right part of Figure 12 by black plain arrows for an instance of `PathClassLoader` and an instance of `DelegateLastClassLoader` (the other class loaders also have this delegate). This delegate is a concept specific to class loaders and has nothing to do with class inheritance. By default, class loaders will delegate to the singleton class `BootClassLoader`, except if a specific class loader is provided when instantiating the new class loader. When a class loader needs to load a class, except for `DelegateLastClassLoader`, it will first ask the delegate, i.e. `BootClassLoader`, and if the delegate does not find the class, the class loader will try to load the class on its own. This behaviour implements a priority and avoids redefining by error a core class of the system, for example, redefining `java.lang.String` that would be loaded by a child class loader instead of its delegates. `DelegateLastClassLoader` behaves slightly differently: it will first delegate to `BootClassLoader`, then it will check its files, and finally, it will delegate to its actual delegate (given when instantiating the `DelegateLastClassLoader`). This behaviour is useful for overriding specific classes of a class loader while keeping the other classes. A normal class loader would prioritise the classes of its delegate over its own.

At runtime, Android instantiates for each application three instances of class loaders described previously: `bootClassLoader`, the unique instance of `BootClassLoader`, and two instances of `PathClassLoader`: `systemClassLoader` and `appClassLoader`. `bootClassLoader` is responsible for loading Android **platform classes**. It is the direct delegate of the two other class loaders instantiated by Android. `appClassLoader` points to the application `.apk` file, and is used to load the classes inside the application `systemClassLoader` is a `PathClassLoader` pointing to `'.'`, the working directory of the application, which is `'/'` by default. The documentation of `ClassLoader.getSystemClassLoader` reports that this class loader is the default context class loader for the main application thread. In reality, the platform classes are loaded by `bootClassLoader` and the classes from the application are loaded from `appClassLoader`. `systemClassLoader` is never used in production according to our careful reading of the AOSP sources.

In addition to the class loaders instantiated by ART when starting an application, the developer of an application can use class loaders explicitly by calling ones from the Android SDK, or by recoding custom class loaders that inherit from the `ClassLoader` class. At this point, accurately modelling the complete class loading algorithm becomes impossible: the developer can program any algorithm of their choice. For this reason, this case is excluded from this chapter, and we focus on the default behaviour where the context class loader is the one pointing to the `.apk` file and where its delegate is `BootClassLoader`. With such a hypothesis, the delegation process can be modelled by the pseudo-code of method `load_class` given in Listing 2.

In addition, it is important to distinguish the two types of platform classes handled by `BootClassLoader` and that both have priority over classes from the application at runtime:

```
def get_mutli_dex_classses_dex_name(index: int):
    if index == 0:
        return "classes.dex"
    else:
        return f"classes{index+1}.dex"

def load_class(class_name: str):
    if is_platform_class(class_name):
        return load_from_boot_class_loader(class_name)
    else:
        index = 0
        dex_file = get_mutli_dex_classses_dex_name(index)
        while file_exists_in_apk(dex_file) and \
            not class_found_in_dex_file(class_name, dex_file):
            index += 1
            dex_file = get_mutli_dex_classses_dex_name(index)
        if file_exists_in_apk(dex_file):
            return load_from_file(dex_file, class_name)
        else:
            raise ClassNotFoundError()
```

Listing 2: Default Class Loading Algorithm for Android Applications

- the ones available in the **Android SDK** (normally visible in the documentation).
- the ones that are internal and that should not be used by the developer. We call them **hidden classes** [25, 34] (not documented).

As a preliminary conclusion, we observe that a priority exists in the class loading mechanism and that an attacker could use it to prioritise an implementation over another one. This could mislead the reverser if they use the one that has the lowest priority. To determine if a class is impacted by the priority given to `BootClassLoader`, we need to obtain the list of classes that are part of Android *i.e.*, the platform classes. We discuss in the next section how to obtain these classes from the emulator.

4.2.3 Determining Platform Classes

Figure 13 shows how classes of Android are used in the development environment and at runtime. In the development environment, Android Studio uses `android.jar` and the specific classes written by the developer. After compilation, only the classes of the developer, and sometimes extra classes computed by Android Studio, are zipped in the APK file, using the multi-dex format. At runtime, the application uses `BootClassLoader` to load the platform classes from Android. Until our work, previous works [25, 34] considered both Android SDK and hidden classes to be in the file `/system/framework/framework.jar` found in the phone itself,

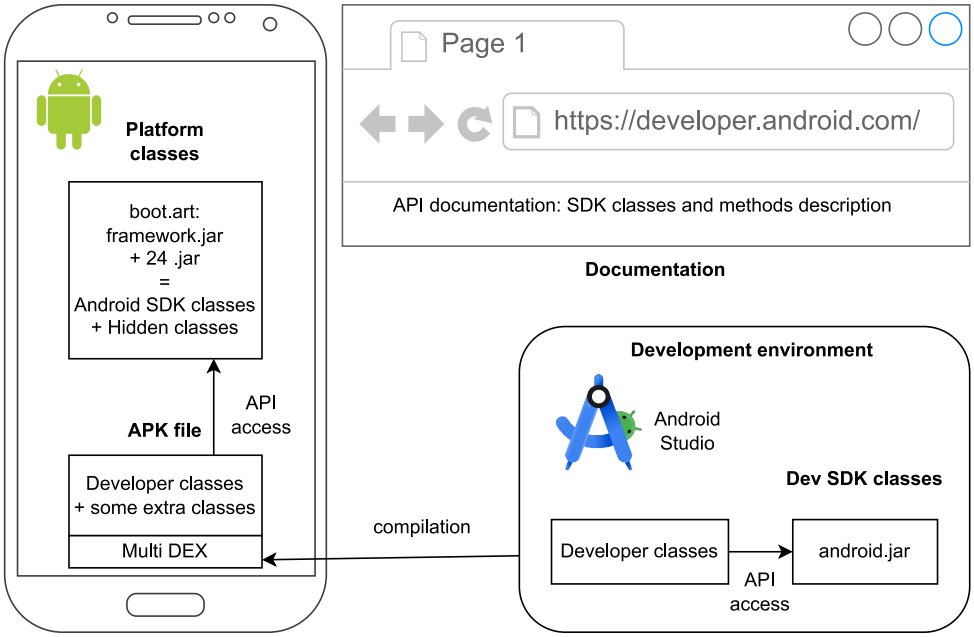


Figure 13: Location of SDK classes during development and at runtime

but we found that the classes loaded by `bootClassLoader` are not all present in `framework.jar`. For example, He *et al.* [25] counted 495 thousand APIs (fields and methods) in Android 12, based on Google documentation on restrictions for non SDK interfaces[1]. However, when looking at the content of `framework.jar`, we only found 333 thousand APIs. Indeed, classes such as `com.android.okhttp.OkHttpClient` are loaded by `bootClassLoader`, listed by Google, but not in `framework.jar`.

For optimisation purposes, classes are now loaded from `boot.art`. This file is used to speed up the start-up time of applications: it stores a dump of the C++ objects representing the **platform classes** (Android SDK and hidden classes) so that they do not need to be generated each time an application starts. Unfortunately, this format is not documented and not retro-compatible between Android versions and is thus difficult to parse. An easier solution to investigate the platform classes is to look at the `BOOTCLASSPATH` environment variable in an emulator. This variable is used to load the classes without the `boot.art` optimisation. We found 25 `.jar` files, including `framework.jar`, in the `BOOTCLASSPATH` of the standard emulator for Android 12 (SDK 32), 31 for Android 13 (SDK 33), and 35 for Android 14 (SDK 35), containing respectively a total of 499 837, 539 236 and 605 098 API methods and fields. Table 7 summarises the discrepancies we found between Google's list and the platform classes we found in Android emulators. Note also that some methods may also be found *only* in the documentation. Our manual investigations

1. https://developer.android.com/guide/app-compatibility/restrictions-non-sdk-interfaces

SDK version	Number of API methods			
	Documented	In emulator	Only documented	Only in emulator
32	495 713	499 837	1060	5184
33	537 427	539 236	1258	3067
34	605 106	605 098	26	18

Table 7: Comparison of API methods between documentation and emulators

suggest that the documentation is not well synchronised with the evolution of the platform classes and that Google has almost solved this issue in API 34.

We conclude that it can be dangerous to trust the documentation and that gathering information from the emulator or phone is the only reliable source. Gathering the precise list of classes and the associated bytecode is not a trivial task.

4.2.4 Multiple DEX Files

For the application class files, Android uses its specific format called DEX: all the classes of an application are loaded from the file `classes.dex`. With the increasing complexity of Android applications, the need arose to load more methods than the DEX format could support in one `.dex` file. To solve this problem, Android started storing classes in multiple files named `classesX.dex` as illustrated by the Listing 3 that generates the filenames read by class loaders. Android starts loading the file `GetMultiDexClassesDexName(0)` (`classes.dex`), then `GetMultiDexClassesDexName(1)` (`classes2.dex`), and continues until finding a value `n` for which `GetMultiDexClassesDexName(n)` does not exist. Even if Android emits a warning message when it finds more than 100 `.dex` files, it will still load any number of `.dex` files that way. This change has the unintended consequence of permitting two classes with the same name but different implementations to be stored in the same `.apk` file using two `.dex` files (*e.g.*, the class `Foo` can be defined both in `classes.dex` and `classes2.dex`).

Android explicitly performs checks that prevent several classes from using the same name inside a `.dex` file. However, this check does not apply to multiple `.dex` files in the same `.apk` file, and a `.dex` can contain a class with a name already used by another class in another `.dex` file of the application. Of course, such a situation should not happen when multiple `.dex` files have been generated properly by Android Studio. Nevertheless, for an attacker controlling the process, this issue raises the question of which class is selected when several classes sharing the same name are present in `.apk` files.

We found that Android loads the class whose implementation is found first when looking in the order of multiple `dexfiles`, as generated by the method `GetMultiDexClassesDexName`. We will

```
std::string DexFileLoader::GetMultiDexClassesDexName(size_t index) {
  return (index == 0) ?
    "classes.dex" :
    StringPrintf("classes%zu.dex", index + 1);
}
```

Listing 3: The method generating the .dex filenames from the AOSP

show later in Section 4.3.2 that this choice is not the most intuitive and can lead to fooling analysis tools when reversing an application. As a conclusion, we model both the multi-dex and delegation behaviours in the pseudo-code of Listing 2.

4.3 Obfuscation Techniques

In this section, we present new obfuscation techniques that take advantage of the complexity of the class loading process. Then, in order to evaluate their efficiency, we reviewed some common Android reverse analysis tools to see how they behave when collisions occur between classes of the APK or between a class of the APK and classes of Android (Android SDK or hidden classes). We call this collision "**class shadowing**", because the attacker's version of the class shadows the one that will be used at runtime. To evaluate if such shadow attacks are working, we handcrafted three applications implementing shadowing techniques to test their impact on static analysis tools. Then, we manually inspected the output of the tools in order to check their consistency with what Android is really doing at runtime. For example, for Apktool, we look at the output disassembled code, and for Flowdroid [8], we check that a flow between `Taint.source()` and `Taint.sink()` is correctly computed.

4.3.1 Obfuscation Techniques

From the results presented in Section 4.2, three approaches can be designed to hide the behaviour of an application.

***Self shadow**: shadowing a class with another from APK* This method consists of hiding the implementation of a class with another one by exploiting the possible collision of class names, as described in Section 4.2.4 with multiple `.dex` files. If reversers or tools ignore the priority order of a multi-dex file, they can take into account the wrong version of a class.

***SDK shadow**: shadowing a SDK class* This method consists of presenting to the reverser a fake implementation of a class of the SDK. This class is embedded in the APK file and has the same name as one of the SDK. Because `BootClassLoader` will give priority to the Android SDK at runtime, the reverser or tool should ignore any version of a class that is contained in the APK. The only constraint when shadowing an SDK class is that the shadowing implementation must respect the signature of real classes. Note that, by introducing a custom class loader, the attacker could invert the priority, but this case is out of the scope of this chapter.

```java
public class Main {
  public static void main(Activity ac) {
    String personal_data = Taint.source();
    String obfuscated_personal_data = Obfuscation.hide_flow(personal_data);
    Taint.sink(ac, obfuscated_personal_data);
  }
}

// customised for each obfuscation technique
public class Obfuscation {
  public static String hide_flow(String personal_data) { ... }
}
```

Listing 4: Main body of test apps

***Hidden shadow**: shadowing a hidden class* This method is similar to the previous one, except the class that is shadowed is a hidden class. Because ART will give priority to the internal version of the class, the version provided in the APK file will be ignored. Such shadow attacks are more difficult to detect by a reverse engineer, who may not know the existence of this specific hidden class in Android.

4.3.2 Impact on Static Analysis Tools

We selected tools that are commonly used to unpack and reverse Android applications. In Chapter 3 (Section 3.2.2), we found only two tools to be still actively maintained: Androguard[1] and Flowdroid[2]. We also noticed that Apktool[3] was a common dependency for a lot of the tools we tested in Chapter 3 (see Table 6), and is still used today. Consequently, we will test the impact of shadow attacks on those three tools. Lastly, because it is a state-of-the-art decompiler for Android applications, we added Jadx[4] to the list of tools we tested.

To evaluate the tools, we designed a single application that we can customise for different tests. Listing 4 shows the main body implementing:

- a possible flow to evaluate FlowDroid: a flow from a method `Taint.source()` to a method `Taint.sink(Activity, String)` through a method `Obfuscation.hide_flow(String)`.
- a possible use of a SDK or hidden class inside the class `Obfuscation` to evaluate platform classes shadowing for other tools.

We used 4 versions of this application:

1. https://github.com/androguard/androguard
2. https://github.com/secure-software-engineering/FlowDroid
3. https://apktool.org/
4. https://github.com/skylot/jadx

1. A control application that does not do anything special: `Obfuscation.hide_flow(String personal_data)` returns `personal_data`. It will be used for checking the expected result of tools.
2. A version that implements self-shadowing: the class `Obfuscation` is duplicated: one is the same as the one in the control app (`Obfuscation.hide_flow(String)` returns its arguments), and the other version returns a constant string. These two versions are embedded in several DEX of a multi-dex application.
3. The third version implements SDK shadowing and needs an existing class of the SDK. We used the SDK class `Pair` as the class to shadow. We put data in a new `Pair` instance and reread the data from the `Pair`. The colliding `Pair` class we created discards the data at the initialisation and stores `null` instead of the argument values. This decoy class breaks the flow of information: Flowdroid will detect the information flow if it uses the actual SDK implementation of `Pair` to compute the DFG, but not if it uses the decoy.
4. The last version tests for Hidden API shadowing. Like for the third one, we similarly store data in `com.android.okhttp.Request` and then retrieve it. Again, the shadowing implementation discards the data.

We used the 4 selected tools on the 4 versions of the application and compared the results on the control application to the results on the other application implementing the different obfuscation techniques. We found that these static analysis tools do not consider the class loading mechanism, either because the tools only look at the content of the application file (*e.g.*, a disassembler) or because they consider class loading to be a dynamic feature and thus out of their scope. In Table 8, we report on the types of shadowing that can trick each tool. A plain circle is a shadow attack that leads to a wrong result. A white circle indicates a tool

Tool	Version	Shadow Attack		
		Self	SDK	Hidden
Jadx	1.5.0	○	●	●
Apktool	2.9.3	○	●	●
Androguard	4.1.2	○	●	●
Flowdroid	2.13.0	●	×	●

●: working
○: works but producing warning or can be seen by the reverser
×: not working

Table 8: Working attacks against static analysis tools

emitting warnings or that displays the two versions of the class. A cross is a tool not impacted by a shadow attack.

4.3.2.1 Jadx

Jadx processes all the classes present in the application, but only saves/displays one class by name, even if two versions are present in multiple `.dex` files. Nevertheless, when multiple classes with the same name are found, Jadx reports it in a comment added to the generated Java source code. This warning stipulates that a possible collision exists and lists the files that contain the different versions of the class. Unfortunately, after reviewing the code of Jadx, we believe that the selection of the displayed class is an undefined behaviour. At least for version 1.5.0 that we tested, we found that Jadx selects the wrong implementation when a class with the same name is present. For example, in `classes2.dex` and `classes3.dex`. We report it with a "○" because warnings are issued.

Shadowing Android SDK and hidden classes is possible in Jadx: there is only one implementation of the class in the application, and Jadx does not have a list of the internal classes of Android: no warning is issued to the reverser that the displayed class is not the one used by Android.

4.3.2.2 Apktool

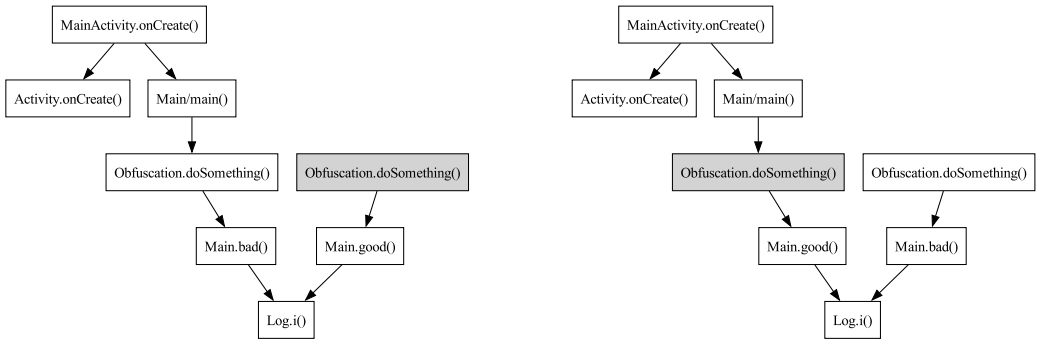
Apktool will store the disassembled classes in a folder that matches the `.dex` file that stores the bytecode. This means that when shadowing a class with two versions in two `.dex` files, the shadow implementations will be disassembled into two directories. No indication is displayed that a collision is possible. It is up to the reverser to have a chance to open the good one.

Similarly to Jadx, using an Android SDK or hidden class will not be detected by the tool that will unpack the fake shadow version.

4.3.2.3 Androguard

Androguard has different usages, with different levels of analysis. The documentation highlights the analysis commands that compute three types of objects: an APK object, a list of DEX objects, and an Analysis object. The APK and the list of `.dex` files are a one-to-one representation of the content of an application, and have the same issues that we discussed with Apktool: they provide the different versions of a shadow class contained in multiple `.dex` files.

The Analysis object is used to compute a method call graph, and we found that this algorithm may choose the wrong version of a shadowed class when using the cross-references that are computed. This leads to an invalid call graph, as shown in Figure 14 b): the two methods `doSomething()` are represented in the graph, but the one linked to `main()` on the graph is the one calling the method `good()` when in fact the method `bad()` is called when running the application.



a) Expected Call Graph b) Call Graph Computed by Androguard

Figure 14: Call Graphs of an application calling `Main.bad()` from a shadowed `Obfuscation` class

Androguard has a method `.is_external()` to detect if the implementation of a class is not provided inside the application and a method `.is_android_api()` to detect if the class is part of the Android API. Regrettably, the documentation of `.is_android_api()` explains that the method is still experimental and just checks a few package names. This means that although those methods are useful, the only indication of the use of an Android SDK or hidden classes is the fact that the class is not in the APK file. Because of that, like for Apktool and Jadx, Androguard has no way to warn the reverser that the shadow of an Android SDK or hidden classes is not the class used when running the application.

4.3.2.4 Flowdroid

We found that when selecting the classes implementation in a multi-dex APK, Soot uses an algorithm close to what ART is performing: Soot sorts the `.dex` bytecode file with a specified `prioritizer` (a comparison function that defines an order for `.dex` files) and selects the first implementation found when iterating over the sorted files. Unfortunately, the `prioritizer` used by Soot is not exactly the same as the one used by the ART. The Soot `prioritizer` will give priority to `classes.dex` and then give priority to files whose name starts with `classes` over other files, and finally will use alphabetical order. This order is good enough for application with a small number of `.dex` files generated by Android Studio, but because it uses the alphabetical order and does not check the exact format used by Android, a malicious developer could hide the implementation of a class in `classes2.dex` by putting a false implementation in `classes0.dex`, `classes1.dex` or `classes12.dex`. Because Flowdroid is based on Soot, it inherits this issue from it.

In addition to self-shadowing, Flowdroid is sensitive to the use of platform classes, as it needs the bytecode of those classes to be able to track data flows. Flowdroid does have a record of SDK classes, and gives priority to the actual SDK classes over the classes implemented in the application, thus defeating SDK shadow attacks. Unfortunately, Flowdroid does not have a record of all platform classes, meaning that using hidden classes breaks the flow tracking. Solving this issue would require finding the bytecode of all the platform classes of the Android version targeted, and, as we said previously, it requires extracting this information from the emulator or phone.

∴

We have seen that tools can be impacted by shadow attacks. In the next section, we will investigate whether these attacks are used in the wild.

4.4 Shadow Attacks in the Wild

In this section, we evaluate in the wild if applications that can be found in the Play Store or other markets use one of the shadow techniques. Our goal is to explore the usage of shadow techniques in real applications. Because we modelled the behaviour of a recent version of Android (SDK 34), we decided not to use our dataset from Chapter 3. The applications in the RASTA dataset span over more than 10 years, and we cannot guarantee that shadow attacks behaved the same during those 10 years. At the very least, self-shadowing would not be possible before the introduction of multi-dex in 2014 – about a fourth of the applications in the RASTA dataset. Instead, we sampled another dataset of recent applications. This way, we can also include malicious applications (in case such techniques would be used to hide malicious code), so we selected 50 000 applications randomly from AndroZoo [3] that appeared in 2023. Malicious applications are spotted in our dataset by using a threshold of 3 over the number of VirusTotal engines reporting an application as malware. This number is provided by Androzoo for scans performed between January 2023 and January 2024, depending on the application. A few applications over the total could not be retrieved or parsed, leading to a final dataset of 49 975 applications. We automatically disassembled the applications to obtain the list of included classes. Then, we check if any shadow attack occurs in the APK itself or with platform classes of SDK 34.

4.4.1 Results

We report in the upper part of Table 9 the statistics about the whole dataset and the three shadow attacks: “self” when a class shadows another one in the APK, “SDK” when a class of the SDK shadows one of the APK, and “Hidden” when a hidden class of Android shadows one of the APK. We observe that, on average, a few classes are shadowed by another class. Note that the median value is 0, meaning that few apps shadow a lot of classes, but the majority of apps do not shadow anything. The number of applications shadowing a hidden API is low,

	Number of apps			Average				Identical Code
		%	% malware	Shadow classes	Median	Target SDK	Min SDK	
For all applications of the dataset								
Self	49 975	100.0%	0.53%	2.1	0	32.1	21.7	74.8%
Sdk	49 975	100.0%	0.53%	6.5	0	32.1	21.7	8.04%
Hidden	49 975	100.0%	0.53%	0.5	0	32.1	21.7	17.42%
Total	49 975	100.0%	0.53%	9	0	32.1	21.7	23.76%
For applications with at least 1 shadow case								
Self	234	0.47%	5.98%	438.1	18	31.4	22.4	74.8%
Sdk	11 755	23.52%	0.38%	27.6	5	32.4	22	8.04%
Hidden	1556	3.11%	0.71%	16.1	1	32.1	22.2	17.42%
Total	12 301	24.61%	0.42%	36.7	6	32.4	22	23.76%

Table 9: Shadow classes compared to SDK 34 for a dataset of 49 975 applications

which is an expected result as these classes should not be known by the developer. We observe a consequent number of applications, 23.52%, that perform SDK shadowing. It can be explained by the fact that some classes that newly appear are embedded in the APK for end users that have old versions of Android: it is suggested by the average value of Min SDK which is 21.7 for the whole dataset: on average, an application can be run inside a smartphone with API 21, which would require to embed all new classes from 22 to 34. This hypothesis about missing classes is further investigated later in this section.

In the bottom part of Table 9, we give the same statistics, but we excluded applications that do not perform any shadowing. For those pairs of shadow classes, we disassembled them using Apktool to perform a comparison using instructions represented in the Smali language. For self-shadow, we compare the pair. For the shadowing of the SDK or Hidden class, we compare the code found in the APK with implementations found in the emulator and `android.jar` of SDK 32, 33, and 34.

Self-shadowing We observe a low number of applications doing self-shadow attacks. For each class that is shadowed, we compared its bytecode with the shadowed one (we compared the Smali instructions generated by Apktool for each method). We observe that 74.8% are identical, which suggests that the compilation process embeds the same class multiple times but makes variations in headers or metadata values. We investigate later in Section 4.4.2 the case of malicious applications.

SDK shadowing For the shadowing of SDK classes, we observe a low ratio of identical classes. This result could lead to the wrong conclusion that developers embed malicious versions of the SDK classes, but our manual investigation shows that the difference is slight and probably due

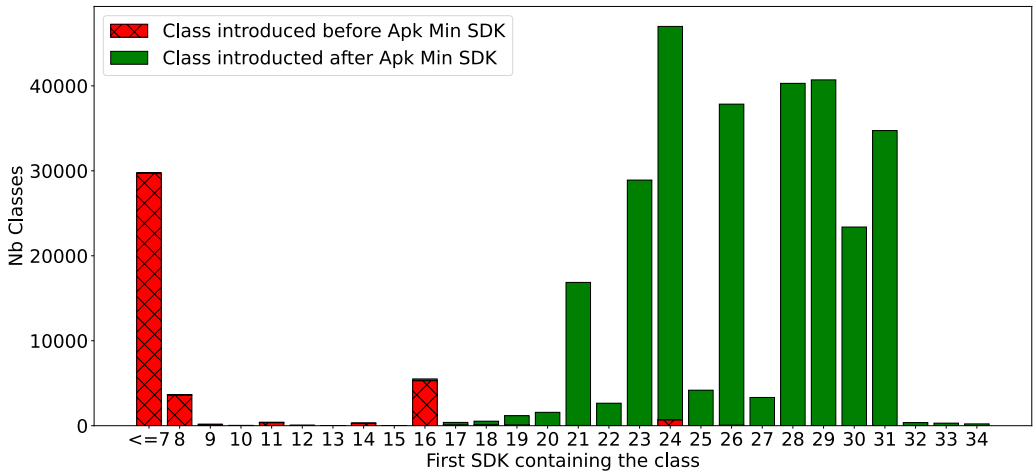


Figure 15: Redefined SDK classes, sorted by the first SDK they appeared in

to compiler optimisation. To go further in the investigation, in Figure 15, we represent these redefined classes with the following rules:

- The class is classified on the X abscissa in the figure according to the SDK it first appeared in.
- The class is counted as "green" (solid) if it first appeared in the SDK **after** the APK min SDK (retro compatibility purpose).
- The class is counted as "red" (hatched) if it first appeared in the SDK **before** the APK min SDK (which is useless for the application as the SDK version is always available).

We observe that the majority of classes are legitimate retro-compatibility additions of classes, especially after SDK 21 (which is the average min SDK, cf. Table 9). Abnormal cases are observed for classes that appeared in API versions 7 and before, 8, and 16. Table 10 reports the top ten classes that shadow the SDK for the three mentioned versions. For SDK before 7, it mainly concerns HTTP classes: for example, the class `HttpParams` is an interface, containing limited bytecode that mostly matches the class already present on the emulator (98.03% of shadowed classes are identical). `HttpConnectionParams`, on the other hand, differs from the platform class, and we observe only 4.99% of identical classes. Manual inspection of some applications revealed that the two main reasons are:

- Instead of checking if the method's attributes are null inline, like Android does, applications use the method `org.apache.http.util.Args.notNull()`. According to comments in the source code of Android[1], the class was forked in 2007 from the Apache 'httpcomponents' project. Looking at the history of the project, the use of `Args.notNull()` was introduced in 2012[2].

This shows that applications are embedding code from more recent versions of this library without realising their version will not be the one used.

- Very small changes that we found can be attributed to the compilation process (e.g. swapping registers: `v0` is used instead of `v1` and `v1` instead of `v0`), but even if we consider them different, they are very similar.

Class	**Occurrences**	**Identical ratio**
redefined for SDK ≤ 7		
`Lorg/apache/http/params/HttpParams;`	1318	98.03%
`Lorg/apache/http/params/HttpConnectionParams;`	1202	4.99%
`Lorg/apache/http/conn/ConnectTimeoutException;`	1200	35.0%
`Lorg/apache/http/params/CoreConnectionPNames;`	1190	99.92%
`Lorg/xmlpull/v1/XmlPullParser;`	1111	52.57%
`Lorg/apache/http/conn/scheme/SocketFactory;`	1074	87.52%
`Lorg/apache/http/conn/scheme/HostNameResolver;`	1072	87.59%
`Lorg/apache/http/conn/scheme/LayeredSocketFactory;`	963	89.41%
`Lorg/json/JSONException;`	945	0.0%
`Lorg/apache/http/conn/ssl/X509HostnameVerifier;`	886	0.79%
redefined for SDK = 8		
`Ljavax/xml/namespace/QName;`	297	0.0%
`Ljavax/xml/namespace/NamespaceContext;`	226	98.23%
`Landroid/net/http/SslError;`	221	31.67%
`Lorg/w3c/dom/UserDataHandler;`	82	92.68%
`Ljavax/xml/transform/TransformerConfigurationException;`	73	69.86%
`Ljavax/xml/transform/TransformerException;`	73	0.0%
`Lorg/w3c/dom/ls/LSException;`	61	63.93%
`Lorg/w3c/dom/TypeInfo;`	54	88.89%
`Lorg/w3c/dom/DOMConfiguration;`	54	46.3%
`Ljavax/xml/transform/TransformerFactoryConfigurationError;`	52	0.0%
redefined for SDK = 16		
`Landroid/annotation/SuppressLint;`	2634	98.48%
`Landroid/annotation/TargetApi;`	2634	98.48%
`Landroid/media/MediaCodec$CryptoException;`	11	18.18%
`Landroid/media/MediaCryptoException;`	10	20.0%
`Landroid/view/accessibility/AccessibilityNodeProvider;`	9	0.0%
`Landroid/view/ActionProvider$VisibilityListener;`	8	12.5%
`Landroid/app/Notification$BigTextStyle;`	7	0.0%
`Landroid/app/Notification$Style;`	7	0.0%
`Landroid/util/LongSparseArray;`	7	0.0%
`Landroid/media/MediaPlayer$TrackInfo;`	7	0.0%

Table 10: Shadow classes compared to SDK 34 for a dataset of 49 975 applications

1. https://cs.android.com/android/platform/superproject/main/+/main:frameworks/base/core/java/org/apache/http/params/HttpConnectionParams.java;drc=3bdd327f8532a79b83f575cc62e8eb09a1f93f3d?

2. https://github.com/apache/httpcomponents-core/commit/9104a92ea79e338d876b1b60f5cd2b243ba7069f?

The remaining 4.99% of classes that are identical to the Android version are classes where the body of the methods is replaced by stubs that throw `RuntimeException("Stub!")`. This code corresponds to what we found in `android.jar`, but not the code we found in the emulator, which is surprising. Nevertheless, we decided to count them as identical, because `android.jar` is the official jar file for developers, and stubs are replaced in the emulator: it is intended by Google developers.

Other results of Table 10 can be similarly discussed: either they are identical with a high ratio, or they are different because of small variations. When substantial differences appear, it is mainly because different versions of the same library have been used or an SDK class is embedded for retro-compatibility.

Hidden shadowing For applications redefining hidden classes, on average, 16.1 classes are redefined (cf bottom part of Table 9). The top 3 packages whose code actually differs from the ones found in Android are `java.util.stream`, `org.ccil.cowan.tagsoup` and `org.json`:

- stream: when looking in more detail, we found that `java.util.stream` was only redefined by 6 applications, but the large number of classes redefined artificially puts the package at the top of the list. It is explained by the fact that developers have included this library, containing a lot of classes that collide with Android.
- tagsoup: `TagSoup` is a library for parsing HTML. Developers do not know that it is part of Android as hidden classes.
- json: there is only one hidden class in `org.json`, redefined by 821 applications: `JSONObject$1`. `org.json` is a package in Android SDK, not a hidden one. However, `JSONObject$1` is an anonymous class not provided by `android.jar` because its class `JSONObject` is an empty stub, and thus, does not use `JSONObject$1`. Thus, this class falls in the category of hidden platform classes.

All these hidden shadow classes are libraries included by the developers who probably did not know that they were already embedded in Android.

4.4.2 Shadowing in Malware Applications

The last column of Table 9 shows the proportion of applications considered as malware because we arbitrarily fixed a threshold of 3 positive detections from VirusTotal reports. For the whole dataset, we have 0.53% of applications considered as malware. We can see that an application that uses self-shadowing is 10 times more likely to be malware, when the proportion of malware among application shadowing platform classes is the same as in the rest of the dataset. Thus, we manually reversed self-shadowing malware and found that the self-shadowing does not look to be voluntary. The colliding classes are often the same implementation, occasionally with minor differences, like different versions of a library. Additionally, we noticed multiple times internal

```java
public class Reflection {
  private static final int ERROR_SET_APPLICATION_FAILED = -20;
  private static final String TAG = "Reflection";
  // ...

  static {
    try {
      Method declaredMethod = Class.class.getDeclaredMethod("forName",
String.class);
      Method declaredMethod2 =
Class.class.getDeclaredMethod("getDeclaredMethod", String.class,
Class[].class);
      Class cls = (Class) declaredMethod.invoke(null,
"dalvik.system.VMRuntime");
      Method method = (Method) declaredMethod2.invoke(cls,
"getRuntime", null);
      setHiddenApiExemptions = (Method) declaredMethod2.invoke(cls,
"setHiddenApiExemptions", new Class[]{String[].class});
      sVmRuntime = method.invoke(null, new Object[0]);
    } catch (Throwable th) { Log.e(TAG, "reflect bootstrap failed:",
th); }
    System.loadLibrary("free-reflection");
    // ...
  }
  // ...
}
```

Listing 5: Implementation of Reflection found in `classes11.dex` (shadows Listing 6)

classes from `com.google.android.gms.ads` colliding with each other, but we believe that it is due to bad processing during the compilation of the application.

The most notable case we found was an application that still exists on the Google Play Store with the same package name[1]. This application contains a self-shadow class `me.weishu.reflection.Reflection` that can be found in Github, in the repository `tiann/FreeReflection`[2]. This class is used to disable Android restrictions on hidden API. At first glance, we believed the shadowing to be done voluntarily for obfuscation purposes. The shadow class that would be seen by a reverser is given in Listing 5: it contains some Java bytecode performing reflection and loading a native library named "free-reflection" (the associated `.so` is missing). The shadowed class that is really executed is summarised in Listing 6. It contains

1. SHA256: `C46A65EA1A797119CCC03C579B61C94FE8161308A3B6A8F55718D6ADAD112546`
2. https://github.com/tiann/FreeReflection

```java
public class Reflection {
  private static final String DEX = "ZGV4CjAzNQCl4EprGS2pXI/
v3OwlBrlfRnX5rmkKVdN0CwAAcA ... AoAAA==";
  private static final String TAG = "Reflection";

  private static native int unsealNative(int i);

  public static int unseal(Context context) {
    return (Build.VERSION.SDK_INT < 28 || BootstrapClass.exemptAll()
|| unsealByDexFile(context)) ? 0 : -1;
  }

  private static boolean unsealByDexFile(Context context) {
    // Decode DEX from base64 and load it as bytecode.
    // ...
  }
  // ...
}
```

Listing 6: Implementation of Reflection executed by ART (shadowed by Listing 5)

a more obfuscated code: a `DEX` field storing base64 encoded DEX bytecode that is later used to load some new code. When looking at this new code stored in the field, we found that it does almost the same thing as the code in the shadow class. Thus, we believe that the developer has upgraded their obfuscation techniques, replacing a native library with inline base64 encoded bytecode. The shadow attack could be unintentional, but it strengthens the masking of the new implementation.

∴

As a conclusion, we observed that:

- SDK shadowing is performed by 23.52% of applications, but is unintentional: these classes are embedded for retro-compatibility purposes or because the developer added a library already present in Android.
- Hidden shadowing rarely occurs and is mainly due to the usage of libraries that Android already contains.
- Malware performs more self-shadowing than goodware applications, and we found a sample where self-shadowing would clearly mislead the reverser.

4.5 Discussion

4.5.1 Countermeasures

Countermeasures against shadow attacks depend on each tool and its objectives. The first important recommendation is to implement the class selection algorithm according to the algorithm described in Listing 2. It should solve any case of self-shadowing, except for tools like Apktool, that do not have to select a class for computing the result, but show the whole application's content. For those tools, a clear warning should be added, pointing out that multiple implementations have been found and displaying the one that will be used at runtime.

Countermeasures against SDK shadow and Hidden shadow attacks are more complex to handle: they require the list of platform classes on the target smartphone and, in some cases, their implementation. The list of SDK classes can be extracted easily from android.jar, but hidden classes need to be obtained by other means. They could be listed directly from the AOSP tree of the Android source code, obtained from Android documentation, or extracted from the phone itself. The first approach requires statically analysing the source code, which can be difficult to achieve as several programming languages are used, and the code base is large and fragmented. Ideally, the documentation would be the best solution, but as discussed earlier in the chapter, it can lack some classes. For this solution to be viable, Google would need to keep the documentation closer to the released version of Android than it currently is. Also, smartphone manufacturers might add additional classes that would not appear in Google documentation. In fact, neither the documentation nor the source code approach can be generalised for all possible versions of Android, as the exact list will depend on the exact targeted device, possibly modified by the manufacturer. Thus, to counter Shadow attacks, the static analysis tools that we evaluated need to embed multiple lists of platform classes, one for each Android version. Then, the best heuristic would be to use the list of platform classes that is closest to the target SDK of the analysed application.

Some tools, like Flowdroid, would require additional countermeasures: to compute the exact flow of data, Flowdroid also needs to analyse the code of platform classes. For the SDK classes, Flowdroid has already analysed them, but the hidden classes have not. In addition to the data flow in hidden classes, Flowdroid needs a list of data sources and sinks from those classes. Other analysis tools may require additional data from platform classes, which may be too difficult to obtain.

We believe that analysis tools can handle shadow attacks to some degree. The implementation of the solution will differ depending on the nature of the tool and may not always require the same implementation effort.

4.5.2 Relation with Obfuscation Techniques

As described in the state of the art, reverse engineers face other techniques of obfuscation, such as packers or native code. These techniques rely on custom class loaders that load new parts of the application from ciphered assets or from the network. The reverse engineers have to study the application dynamically to recover new classes and eventually go back to a static phase to understand the behaviour of the application. In this section, we compare shadow attacks with these techniques and discuss how they interact with them.

Advanced obfuscation techniques relying on packers have a higher impact on the difficulty of performing a static analysis compared to shadow attacks. Most of the time, the reverse engineer cannot deobfuscate the application without performing a dynamic analysis. For these reasons, approaches have been designed to assist the capture of the bytecode that is loaded dynamically, after the precise time where the deobfuscation methods have been executed [68, 70, 74]. On the contrary, a shadow attack can be easily defeated by implementing our algorithm in the static analysis tool, as discussed earlier in Section 4.5.1. Nevertheless, shadow attacks are stealthier than packers or native code. Packers can be easily spotted by artefacts left behind in the application or by detecting classes implementing a custom class loading mechanism. On the contrary, an extra class implementing a shadow attack, that would not be executed, could contain voluntarily little code, compared to the executed class of Android. Such an attack would be more discreet than a packer that adds in the application a lot of possibly native code.

Combining regular obfuscation techniques with shadow attacks can be achieved in two ways.

First, the attacker could hide the code of a packer or a native call by using a shadow attack. For example, by colliding a class of the SDK, a control flow analysis could be wrongly computed, leading to considering that part of the code to be dead, which would mislead the reverse engineer about the use of this part that contains a packer. At runtime, this code would be triggered, unpacking new code.

Second, the attacker could use a packer to unpack code at runtime in a first phase. The reverse engineer would have to perform a dynamic analysis, for example, using a tool such as Dexhunter [74], to recover new DEX files that are loaded by a custom class loader. Then, the reverse engineer would go back to a new static analysis and could have the problem of solving shadow attacks, for example, if a class is defined multiple times in the loaded DEX files.

Because the interactions between shadow attacks and other obfuscation techniques often rely on a loading mechanism implemented by the developer, investigating these cases requires analysing the Java bytecode that is handling the loading. This problem is left as future work.

4.5.3 Limitations

During the analysis of the ART internals, we made the hypothesis that its different operating modes are equivalent: we analysed the loading process for classes stored as non-optimised `.dex` format, and not for the pre-compiled `.oat`. It is a reasonable hypothesis to suppose that the two implementations have been produced from the same algorithm using two compilation workflows. Similarly, we assumed that the platform classes stored in `boot.art` are the same as the ones in `BOOTCLASSPATH`. We confirm empirically our hypothesis on an Android Emulator, but we may have missed some edge cases.

The comparison of Smali code can lead to underestimated values, for example, if the compilation process performs minor modifications such as instruction reordering. The ratios reported in this study for the comparison of code are thus a lower bound and would be higher with a more precise comparison. In addition, platform classes are stored differently in older versions of Android and cannot be easily retrieved. For this reason, we did not compare the classes found in applications to their versions older than SDK 32 to avoid producing unreliable statistics for those versions.

4.5.4 Future Works

As we just said, our Smali-based comparison of class implementation is quite naive and could use more work. It could be insightful to be able to detect exactly when two classes are from the same source file, or which version of a library a class belong to. More importantly, a better comparison technique would allow us to detect cases where the shadowed library has actual malicious bytecode added that we could have missed manually.

Additionally, the question of dynamic class loaders, used manually by the application developer, is interesting. This is reaching the limits of static analysis; those cases involve dynamically loading bytecode, and in many cases, the classes loaded by those class loaders are not even available for analysis. However, even with dynamic analysis, the behaviour of class loaders can still be an issue, especially when the analysis is performed by alternating static and dynamic analysis, as is often the case when manually reversing an application. To handle those cases, it could be interesting to develop a method to model any arbitrary class loader, either by analysing its bytecode or by interacting with an instance of the class loader dynamically.

In September 2024 (just after we finished this work), Android 15 introduced support for the new version 41 of the DEX format. We can expect this version of DEX to become the norm in a few years. The most notable change in version 41 is the new container format: instead of storing the bytecode in separate DEX files, the different files can now be concatenated into one unique file. There is also some permeability between the concatenated files: some structures stored in one file can be used by the next concatenated files. This significant change in the bytecode storage is similar to the introduction of the multi-dex format. Considering that self-shadowing

is only possible because of the multi-dex format, we can expect this change to have the potential to introduce new, similar issues. Thus, we believe that the implementation details of this new version should be studied and modelled properly to avoid introducing new issues when updating analysis tools to support it. Just by reading the specification[1], we believe that self-shadowing between concatenated DEX files is possible, unless additional checks are enforced by the ART when loading the file.

4.6 Conclusion

This chapter has presented three shadow attacks that allow malware developers to fool static analysis tools when reversing an Android application. By including multiple classes with the same name or by using the same name as a class of the Android SDK, the developer can mislead a reverse engineer or impact the result of a flow analysis, such as those of Androguard or Flowdroid.

We explored whether such shadow attacks are present in a dataset of 49 975 applications. We found that on average, 23.52% of applications are shadowing the SDK, mainly for retro-compatibility purposes and library embedding. More suspiciously, 3.11% of applications are shadowing a hidden class, which could lead to unexpected execution as these classes can appear/ disappear with the evolution of Android internals. Investigations for applications that defined classes multiple times suggest that the compilation process or the inclusion of different versions of the same library is the main explanation. Finally, when investigating malware samples, we found a specific sample containing a shadow attack that would hide a part of the critical code from a reverse engineer studying the application.

1. https://source.android.com/docs/core/runtime/dex-format#container

Pb2: *What is the default Android class loading algorithm, and does it impact static analysis?*

Listing 2 model the class loading algorithm: platform classes have priority over classes stored in `classes.dex` which have priority over `classes<n>.dex` (where $n \in [\![2, +\infty[\![$ and $\forall i \in [\![2, n[\![, \exists$ `classes<i>.dex`) which has priority over `classes<n+1>.dex`.

Failing to implement this model (*i.e.*, by ignoring some platform classes or by sorting the `classes<n>.dex` alphabetically instead of numerically) can cause static analysis tools to compute an incorrect representation of the analysed application.

Chapter 5

The Application of Theseus: After Adding Runtime Data, it is Still Your Application

---

Despite everything, it's still you.

— Undertale, Toby Fox

Some applications use dynamic code loading and reflection calls that prevent static analysis tools from analysing the complete application. This can be detected with dynamic analysis; however, the collected data is not enough to analyse the application further: most tools do not have a way to process this additional data. In this chapter, we propose to use dynamic analysis to collect information related to dynamic code loading and reflection, and to encode this information in the bytecode of the application to allow further analysis. We compared the results of analysis on applications before and after the transformation, using tools like Flowdroid or Androguard, and found that the additional information is indeed processed by the static analysis tools. We also compared the finishing rate of the tools, using the same experiment as in Chapter 3, and found that the finishing rate is generally only slightly negatively impacted by the transformation.

5.1 Introduction

In the previous chapter, we studied the static impact of class loaders. However, as we focused on the default behaviour of Android, we ignored the main use of class loaders for developers: dynamic code loading. In this chapter, we address this issue, as well as the issue of reflection that often accompanies dynamic code loading. Dynamic code loading is the practice of loading at runtime bytecode that was not already part of the original bytecode of the application. This bytecode can be stored as assets of the application, downloaded from a remote server, or even generated algorithmically by the application. This is a problem for analysis: when the bytecode is not already visible in the application, it cannot be analysed statically. Meanwhile, reflection is the action of using code to manipulate objects representing structures of the code itself, like classes or methods. The main issue for analysis occurs when it is used to call methods. A static analysis will show calls to `Method.invoke()`, but not the actual method invoked.

In both cases, static analysis falls short, as the information to analyse may be generated just in time for its use. For such cases, dynamic analysis is a more appropriate approach. It can be used to collect the missing information while the application is running. However, having this information does not mean that the application can now be analysed in its entirety. Generic analysis tools rarely have an easy way to read additional information about an application before analysing, and when they do, it is not standard. The usual approach for hybrid analysis (analyses that mix static and dynamic analysis) is to select one specific static tool and modify its code to take into account the additional data collected by dynamic analysis. This limits the reverse engineer to a few tools that they took the time to study and modify for the task. In this chapter, we propose to modify the code of the application to add the information needed for analysis in a format that any analysis tool can use. This way, the analyst is no longer limited in their choice of tool and can focus on the actual analysis of the application.

We structured this chapter as follows: We first present an overview of our method in Section 5.2. We then present the transformations we apply to the application in Section 5.3 and the dynamic analysis we perform in Section 5.4. In Section 5.5 compare the results of different tools on the initial application versus the modified application. To complete this chapter, in Section 5.6 we discuss the limits of our solution, as well as directions for future work. Finally, we conclude in Section 5.7.

5.2 Overview

Our objective is to make available some dynamic information to any analysis tool able to analyse an Android APK. To do so, we elected to follow the same approach as a few contributions we presented in Chapter 2, such as DroidRA [32], and use instrumentation. As a reminder, DroidRA is a tool that uses COAL to compute reflection data statically, then instruments the application to directly call the methods. Contrary to DroidRA, we chose to use dynamic analysis. This

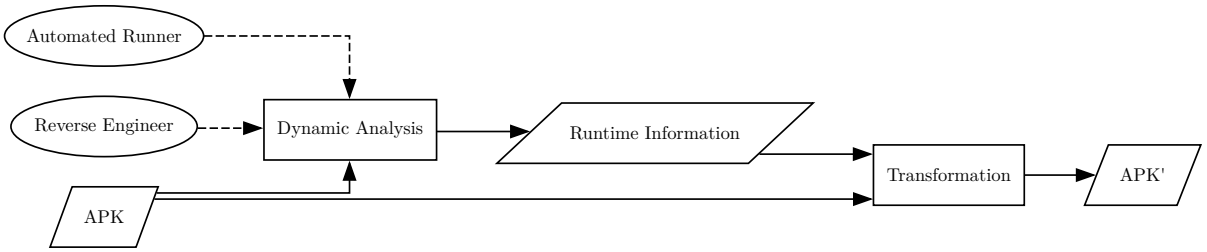


Figure 16: Process to add runtime information to an APK

allows us to collect information that is simply not available statically (*e.g.*, a string sent from a remote command and control server). The tradeoff here is the lack of exhaustiveness: dynamic analysis is known to have code coverage issues.

Figure 16 summarises our process. We first take an application that we analyse dynamically. To improve code coverage, either a reverse engineer or an automated runner will interact with the application. During this analysis, we use Frida to capture dynamic information like the names of the methods called using reflection and bytecode loaded at runtime. This analysis is described in Section 5.4.

The data collected by this analysis is then combined with the application, transforming the application into another one that can then be analysed further. We present the details of this transformation in Section 5.3. Since the transformation drives the data we need to collect, we have decided to place this section first in this chapter.

5.3 Code Transformation

In this section, we will see how we can transform the application code to make dynamic code loading and reflective calls more analysable by static analysis tools.

5.3.1 Transforming Reflection

In Android, reflection allows applications to instantiate a class or call a method without having this class or method appear in the bytecode. Instead, the bytecode uses the generic classes `Class`, `Method` and `Constructor`, which represent any existing class, method or constructor. Reflection often starts by retrieving the `Class` object representing the class to use. This class is usually retrieved using a `ClassLoader` object (though there are other ways to get it). Once the class is retrieved, it can be instanciated using the deprecated method `Class.newInstance()`, as shown in Listing 7, or a specific method can be retrieved. The current approach to instantiate a class is to retrieve the specific `Constructor` object, then call `Constructor.newInstance(..)` like in Listing 8. Similarly, to call a method, the `Method` object must be retrieved, then called using `Method.invoke(..)`, as shown in Listing 9.

```
1 ClassLoader cl = MainActivity.class.getClassLoader();
2 Class clz = cl.loadClass("com.example.Reflectee");
3 Object obj = clz.newInstance();
```

Listing 7: Instantiating a class using `Class.newInstance()`

```
1 Constructor cst = clz.getDeclaredConstructor(String.class);
2 Object obj = cst.newInstance("Hello Void");
```

Listing 8: Instantiating a class using `Constructor.newInstance(..)`

```
1 Method mth = clz.getMethod("myMethod", String.class);
2 Object[] args = {(Object)"an argument"};
3 String retData = (String) mth.invoke(obj, args);
```

Listing 9: Calling a method using reflection

Although the process seems to differ between class instantiation and method call from the Java standpoint, the runtime operations are very similar. When instantiating an object with `Object obj = cst.newInstance("Hello Void")`, the constructor method `<init>(Ljava/lang/String;)V`, represented by the `Constructor cst`, is called on the object `obj`. Thus, even for instantiation, a method is called at some point.

One of the main reasons to use reflection is to access classes that are neither platform classes nor in the application bytecode, as is often the case when dealing with classes from dynamically loaded bytecode. Indeed, if the ART were to encounter an instruction referencing a class that cannot be loaded by the current class loaded, it would crash the application.

To allow static analysis tools to analyse an application that uses reflection, we want to replace the reflection call with a bytecode chunk that actually calls the method and can be analysed by any static analysis tool. In Section 5.3.2, we deal with the issue of dynamic code loading so that the classes used are, in fact, present in the application.

A notable issue is that a specific reflection call can call different methods. Listing 10 illustrates a worst-case scenario where any method can be called at the same reflection call. In those situations, we cannot guarantee that we know all the methods that can be called (*e.g.*, the name of the method called could be retrieved from a remote server). In addition, the method we propose in Section 5.4 is a best effort approach to collect reflection data: like any dynamic analysis, it is limited by its code coverage.

To handle those situations, instead of entirely removing the reflection call, we can modify the application code to test if the `Method` (or `Constructor`) object matches any of the methods

```
1 Object myInvoke(Object obj, Method mth, Object[] args) throws .. {
2   return mth.invoke(obj, args);
3 }
```

Listing 10: A reflection call that can call any method

observed dynamically, and if so, directly call the method. If the object does not match any expected method, the code can fall back to the original reflection call. DroidRA [32] has a similar solution, except that reflective calls are always evaluated, and the static equivalent follows just after, guarded behind an opaque predicate that is always false at runtime. Listing 11 demonstrates this transformation for the code originally in Listing 9. Let's suppose that we observed dynamically a call to a method `Reflectee.myMethod(String)` at line 3 when monitoring the execution of the code of Listing 9. In Listing 11, at line 25, the `Method` object `mth` is checked using a method we generated and injected in the application (defined at line 2 in the listing). This method checks if the method name (line 5), its parameters (lines 6-9), its return type (lines 10-11) and its declaring class (lines 13-14) match the expected method. If it is the case, the method is used directly (line 26) after casting the arguments and associated object into the types/classes we just checked. If the check line 25 does not pass, the original reflective call is made (line 28). If we were to expect other possible methods to be called in addition to `myMethod`, we would add `else if` blocks between lines 26 and 27, with other check methods reflecting each potential method call.

The check of the `Method` value is done in a separate method injected inside the application to avoid cluttering the application too much. Because Java (and thus Android) uses polymorphic methods, we cannot just check the method name and its class, but also the whole method signature. We chose to limit the transformation to the specific instruction that calls `Method.invoke(..)`. This drastically reduces the risks of breaking the application, but leads to a lot of type casting. Indeed, the reflection call uses the generic `Object` class, but actual methods usually use specific classes (*e.g.*, `String`, `Context`, `Reflectee`) or scalar types (*e.g.*, `int`, `long`, `boolean`). This means that the method parameters and object on which the method is called must be downcasted to their actual type before calling the method, then the returned value must be upcasted back to an `Object`. Scalar types especially require special attention. Java (and Android) distinguish between scalar types and classes, and they cannot be mixed: a scalar cannot be cast into an `Object`. However, each scalar type has an associated class that can be used when doing reflection. For example, the scalar type `int` is associated with the class `Integer`, the method `Integer.valueOf()` can convert an `int` scalar to an `Integer` object, and the method `Integer.intValue()` converts back an `Integer` object to an `int` scalar. Each time the method called by reflection uses scalars, the scalar-object conversion must be made before

```
class T {
  static boolean check_is_reflectee_mymethod_e398(Method mth) {
    Class<?>[] paramTys = mth.getParameterTypes();
    return (
      meth.getName().equals("myMethod") &&
      paramTys.length == 1 &&
      paramTys[0].descriptorString().equals(
        String.class.descriptorString()
      ) &&
      mth.getReturnType().descriptorString().equals(
        String.class.descriptorString()
      ) &&
      mth.getDeclaringClass().descriptorString().equals(
        Reflectee.class.descriptorString()
      )
    )
  }
}

...

Method mth = clz.getMethod("myMethod", String.class);
Object[] args = {(Object)"an argument"}
Object objRet;
if (T.check_is_reflectee_mymethod_e398abf7d3ce6ede(mth)) {
  objRet = (Object) ((Reflectee) obj).myMethod((String)args[0]);
} else {
  objRet = mth.invoke(obj, args);
}
String retData = (String) objRet;
```

Listing 11: Listing 9 after the de-reflection transformation

calling it. And finally, because the instruction following the reflection call expects an `Object`, the return value of the method must be cast into an `Object`.

This back and forth between types might confuse some analysis tools. This could be improved in future works by analysing the code around the reflection call. For example, if the result of the reflection call is immediately cast into the expected type (*e.g.*, in Listing 9, the result is cast to a `String`), there should be no need to cast it to Object in between. Similarly, it is common to have the method parameter arrays generated just before the reflection call and never be used again (This is due to `Method.invoke(..)` being a varargs method: the array can be generated

by the compiler at compile time). In those cases, the parameters could be used directly without the detour inside an array.

5.3.2 Transforming Code Loading (or Not)

An application can dynamically import code from several formats like DEX, APK, JAR or OAT, either stored in memory or in a file. Because it is an internal, platform-dependent format, we elected to ignore the OAT format. Practically, JAR and APK files are zip files containing DEX files. This means that we only need to find a way to integrate DEX files into the application.

We saw in Chapter 4 the class loading model of Android. When doing dynamic code loading, an application defines a new `ClassLoader` that handles the new bytecode, and starts accessing its classes using reflection. We also saw in Chapter 4 that Android now uses the multi-dex format, allowing it to handle any number of DEX files in one class loader. Therefore, the simpler way to give access to the dynamically loaded code to static analysis tools is to add the dex files in the application as additional multi-dex bytecode files. This should not impact the class loading model as long as there is no class collision (we will explore this in Section 5.3.3) and as long as the original application does not try to access inaccessible classes (we will develop this issue in Section 5.6).

In the end, we decided **not** to modify the original code that loads the bytecode. Most tools already ignore dynamic code loading, and, with the dynamically loaded bytecode added using the multi-dex format, they already have access to it. At runtime, although the bytecode is already present in the application, the application will still dynamically load the code. This ensures that the application keeps working as intended, even if the transformation we applied is incomplete. Specifically, to call dynamically loaded code, an application needs to use reflection, and we saw in Section 5.3.1 that we need to keep reflection calls, and in order to keep reflection calls, we need the class loader created when loading bytecode.

To summarise, we do not modify the existing bytecode. Instead, we add the intercepted bytecode to the application as additional DEX files using the multi-dex format, as represented in Figure 17.

5.3.3 Class Collisions

We saw in Chapter 4 that having several classes with the same name in the same application can be problematic. In Section 5.3.2, we are adding new code. By doing so, we increase the probability of having class collisions: The developer may have reused a helper class in both the dynamically loaded bytecode and the application, or an obfuscation process may have renamed classes without checking for intersection between the two sources of bytecode. When loaded dynamically, the classes are in a different class loader, and the class resolution is resolved at runtime, like we saw in Section 4.2. We decided to restrain our scope to the use of class loaders

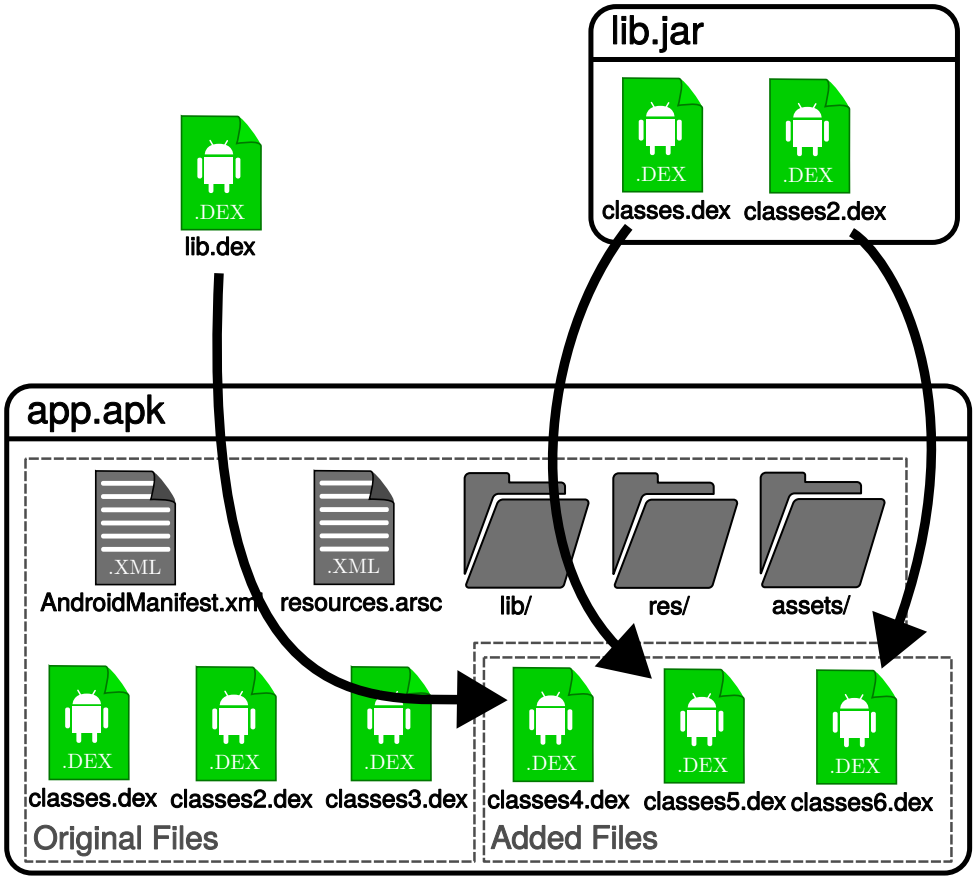


Figure 17: Inserting DEX files inside an APK

from the Android SDK. In the absence of class collision, those class loaders behave seamlessly and adding the classes to the application maintains the behaviour.

When we detect a collision, we rename one of the colliding classes in order to be able to differentiate between classes. To avoid breaking the application, we then need to rename all references to this specific class and be careful not to modify references to the other class. To do so, we regroup each class by the class loaders that define them. Then, for each colliding class name and each class loader, we check the actual class used by the class loader. If the class has been renamed, we rename all references to this class in the classes defined by this class loader. To find the class used by a class loader, we reproduce the behaviour of the different class loaders of the Android SDK. This is an important step: remember that the delegation process can lead to situations where the class defined by a class loader is not the class that will be loaded when querying the class loader. The pseudo-code in Listing 12 shows the three steps of this algorithm:

```
defined_classes = set()
redifined_classes = set()

# Rename the definition of redifined classes
for cl in class_loaders:
  for clz in defined_classes.intersection(cl.defined_classes):
    cl.rename_definition(clz)
    redifined_classes.add(clz)
  defined_classes.update(cl.defined_classes)

# Rename reference of redifined classes
for cl in class_loaders:
  for clz in redifined_classes:
    defining_cl = cl.resolve_class(clz).class_loader
    cl.rename_reference(clz, defining_cl.new_name(clz))

# Merge the class loader into a flat APK
new_apk = Apk()
for cl in class_loaders:
  for dex in cl.get_dex():
    new_apk.add_dex(dex)
```

Listing 12: Pseudo-code of the renaming algorithm

- First, we detect collisions and rename class definitions to remove the collisions.
- Then we rename the reference to the colliding classes to make sure the right classes are called.
- Ultimately, we merge the modified DEX files of each class loader into one Android application.

5.3.4 Implementation Details

Our initial idea was to use Apktool, but in Chapter 3, we found that many errors raised by tools were due to trying to parse Smali incorrectly. Thus, we decided to avoid Apktool.

Most of the contributions of the state of the art that perform instrumentation rely on Soot. Soot works on an intermediate representation, Jimple, that is easier to manipulate. However, Soot can be cumbersome to set up and use, and we initially wanted better control over the modified bytecode. In addition, although it might be due to the fact that they performed more complex analysis, tools based on Soot showed a trend of consuming a lot of memory and failing with unclear errors, supporting us in our idea of avoiding Soot. For these reasons, we decided to make our own instrumentation library from scratch.

That library, Androscalpel, requires being able to parse, modify and generate valid DEX files. It was not as difficult as one would expect, thanks to the clear documentation of the Dalvik

format from Google[1]. In addition, when we had doubts about the specification, we had the option to check the implementation used by Apktool[2], or the code used by Android to check the integrity of the DEX files[3].

We chose to use Rust to implement this library. It has both good performance and ergonomics. For instance, we could parallelise the parsing and generation of DEX files without much effort. Because we are not using a high-level intermediate language like Jimple (used by Soot), the management of the Dalvik registers in the methods has to be done manually (by the user of the library), the same way it has to be done when using Apktool. This poses a few challenges.

A method declares a number of internal registers it will use (let's call this number n), and has access to an additional number of registers used to store the parameters (let's call this number p). Each register is referred to by a number from 0 to 65535. The internal registers are numbered from 0 to $n-1$, and the parameter registers from n to $n+p-1$. This means that when adding new registers to the method when instrumenting it (let's say we want to add k registers), the new registers will be numbered from n to $n+k-1$, and the parameter registers will be renumbered from $[\![n, n+p[\![$ to $[\![n+k, n+k+p[\![$. In general, this is not an issue, but some instructions can only operate on some registers (*e.g.*, `array-length`, which stores the length of an array in a register, only works on registers numbered between 0 and 8 excluded). This means that adding registers to a method can be enough to break a method. We solved this by adding instructions that move the content of registers $[\![n+k, n+k+p[\![$ to the registers $[\![n, n+p[\![$, and keeping the original register numbers ($[\![n, n+p[\![$) for the parameters in the rest of the body of the method.

The next challenge arises when we need to use one of the new registers with an instruction that only accepts registers lower than $n+p$. In such cases, a lower register must be used, and its content will be temporarily saved in one of the new registers. This is not as easy as it seems: the Dalvik instructions differ depending on whether the register stores a reference or a scalar value, and Android does check that the register types match the instructions. The type of the register can be computed from the control flow graph of the method (we added the computation of such a graph, with the type of each register, as a feature in Androscalpel). An edge case that must not be overlooked is that each instruction inside a `try` block is branching to each of the `catch` blocks. This is a problem: it prevents us from restoring the registers to their original values before entering the `catch` blocks (or, if we restore the values at the beginning of the `catch` blocks and an exception is raised before the value is saved, the register will be overwritten by

1. https://source.android.com/docs/core/runtime/dex-format
2. https://github.com/JesusFreke/smali
3. https://cs.android.com/android/platform/superproject/main/+/main:art/libdexfile/dex/dex_file_verifier.cc;drc=11bd0da6cfa3fa40bc61deae0ad1e6ba230b0954

an invalid value). This means that when modifying the content of a `try` block, the block must be split into several blocks to prevent impromptu branching.

One thing we noticed when manually instrumenting applications with Apktool is that sometimes the repackaged applications cannot be installed or run due to some files being stored incorrectly in the new application (*e.g.*, native library files must not be compressed). We also found that some applications deliberately store files with names that will crash the zip library used by Apktool. For this reason, we also used our own library to modify the APK files. We took special care to process the least possible files in the APKs, and only strip the DEX files and signatures, before adding the new modified DEX files at the end.

Unfortunately, we did not have time to compare the robustness of our solution to existing tools like Apktool and Soot, but we did a quick performance comparison, summarised in Section 5.5.5. In hindsight, we probably should have taken the time to find a way to use smali/backsmali (the backend of Apktool) as a library or use SootUp to do the instrumentation, but neither option has documentation to instrument applications this way. At the time of writing, the feature is still being developed, but in the future, Androguard might also become an option to modify DEX files. Nevertheless, we published our instrumentation library, Androscalpel, for anyone who wants to use it (see Appendix A).

∴

Now that we saw the transformations we want to make, we know the runtime information we need to do it. In the next section, we will propose a solution to collect that information.

5.4 Collecting Runtime Information

To perform the transformations described in Section 5.3, we need information like the name and signature of the method called with reflection, or the actual bytecode loaded dynamically. We decided to collect that information through dynamic analysis. We saw in Chapter 2 different contributions that collect this kind of information. In the end, we decided to keep the analysis as simple as possible, so we avoided using a custom Android build like DexHunter and instead used Frida to instrument the application and intercept calls to the methods of interest. Section 5.4.1 presents our approach to collect dynamically loaded bytecode, and Section 5.4.2 presents our approach to collect the reflection data. Because using dynamic analysis raises the concern of coverage, we also need some interaction with the graphical user interface of the application during the analysis. Ideally, a reverse engineer would do the interaction. Because we wanted to analyse many applications in a reasonable time, we replaced this engineer with an automated runner that simulates the interactions. We discuss this option in Section 5.4.3.

5.4.1 Collecting the Dynamically Loaded Bytecode

Initially, we considered instrumenting the constructor methods of the class loaders of the Android SDK. However, this is a significant number of methods to instrument, and looking at older applications, we realised that we missed the `DexFile` class. `DexFile` is now deprecated but still usable class that can be used to load bytecode dynamically. We initially missed this class because it is neither a `ClassLoader` class nor an SDK class (anymore). To avoid running into this kind of oversight again, we decided to look at the ART source code and list all the places where the internal functions used to parse bytecode are called. We found that all those calls are from under either `DexFile.openInMemoryDexFilesNative(..)` or `DexFile.openDexFileNative(..)`, two hidden API methods. As a reference, in 2015, DexHunter [74] already noticed `DexFile.openDexFileNative(..)` (although in the end DexHunter intruments another function, `DefineClass(..)`). `DefineClass(..)` is still a good function to instrument, but it is a C++ native method that does not have a Java interface, making it harder to work with using Frida, and we want to avoid patching the source code of the ART like DexHunter did. For this reason, we decided to hook `DexFile.openInMemoryDexFilesNative(..)` and `DexFile.openDexFileNative(..)` instead. Those methods take a list of Android code files as argument, either in the form of in-memory byte arrays or file paths, and a reference to the classloader associated with the code. Instrumenting those methods allows us to collect all the code files loaded by the ART and associate them with their class loaders.

5.4.2 Collecting Reflection Data

As described in Section 5.3.1, there are 3 methods that we need to instrument to capture reflection calls: `Class.newInstance()`, `Constructor.newInstance(..)` and `Method.invoke(..)`. Because Java has polymorphism, we need not only the method name and defining class, but also the whole signature of the method. In addition to that, in case there are several classes with the same name as the defining class, we also need the classloader of the defining class to distinguish it from the other classes.

Where the reflection method is called is more difficult to find. In order to correctly modify the application, we need to know which specific call to a reflection method we intercepted. Specifically, we need the caller method (once again, we need the method name, full signature, defining class and its classloader), and the exact instruction that called the reflection method (in case the caller method uses reflection several times in different sites). This information is more difficult to collect than one would expect. It is stored in the stack, but before the SDK 34, the stack was not directly accessible programmatically. Historically, when a reverse engineer needed to access the stack, they would trigger and catch an exception and get the stack from that exception. The issue with this approach is that data stored in exceptions is meant for debugging. In particular, the location of the call in the bytecode has a different meaning depending on the debug information encoded in the bytecode. It can either be the address of the bytecode

instruction invoking the callee method in the instruction array of the caller method, or the line number of the original source code that calls the callee method. Fortunately, in the SDK 34, Android introduced the `StackWalker` API. This API allows to programatically travel the current stack and retrieve information from it, including the bytecode address of the instruction calling the callee methods. Considering that the line number is not a reliable information, we chose to use the new API, despite the restrictions that come with choosing such a recent Android version (it was released in October 2023, around 2 years ago, and less than 50% of the current Android market share supports this API today[1]).

5.4.3 Application Execution

Dynamic analysis requires actually running the application. In order to test multiple applications automatically, we needed to simulate human interactions with the applications. In Chapter 2, we presented a few solutions to explore an application dynamically. We first eliminated Sapienz [41], as it relies on an application instrumentation library called ELLA, which has not been updated for 9 years. We also chose to avoid the Monkey because we noticed that it often triggers events that close the application (events like pressing the 'home' button, or opening the general settings drop-down menu at the top of the screen). Stoat [60] and GroddDroid [1] use UI Automator to interact with the application. UI Automator is a standard Android API intended for automatic testing. Both Soat and GroddDroid perform additional analysis on the application to improve the exploration. In the end, we elected to use the most basic execution mode of GroddDroid that does not need this additional analysis. It explores the application following a depth-first search algorithm. We chose this option to keep the exploration lightweight and limit the chance of crashing the analysis (we saw in Chapter 3 the issues brought by complex analysis). It might be interesting in future work to explore more advanced exploration techniques.

Because we are using Frida, we do not need to use a custom version of Android with a modified ART or kernel. However, we decided not to inject Frida into the original application. This means we need to have root access to directly run Frida in Android, which is not a normal thing to have on Android. Because dynamic analysis can be slow, we also decided to run the applications on emulators. This makes it easier to run several analyses in parallel. The alternative would have been to run the application on actual smartphones, and would have required multiple phones to run the analysis in parallel. For simplicity, we chose to use Google's Android emulator for our experiment. We spawned multiple emulators, installed Frida on them, took a snapshot of the emulator before installing the application to analyse. Then we run the application for five minutes with GroddRunner, and at the end of the analysis, we reload the snapshot in case the

1. https://gs.statcounter.com/android-version-market-share/mobile-tablet/worldwide/#monthly-202401-202508

application modified the system in some unforeseen way. If at some point an emulator stops responding for too long, we terminate it and restart it.

As we will see in Section 5.5.1, our experimental setup is quite naive and still requires improvement. For example, we do not implement any anti-evasion techniques, which can be a significant issue when analysing malware. Nonetheless, the benefit of our implementation is that it only requires an ADB connection to a phone with a rooted Android system to work. Of course, to analyse a specific application, a reverse engineer could use an actual smartphone and explore the application manually. It would be a lot more stable than our automated batch analysis setup.

∴

Now that we saw both the dynamic analysis setup and the transformation we want to perform on the APKs, we put our proposed approach into practice. In the next section, we will run our dynamic analysis on APKs and study the data collected, as well as the impact the instrumentation has on applications and different analysis tools.

5.5 Results

To study the impact of our transformation on analysis tools, we reused applications from the dataset we sampled in Chapter 3. Because we are running the application on a recent version of Android (SDK 34), we only took the most recent applications: the one collected in 2023. This represents 5000 applications over the 62 525 total of the initial dataset. Among them, we could not retrieve 43 from Androzoo, leaving us with 4957 applications to test.

We will first look at the results of the dynamic analysis and look at the bytecode we intercepted. Then, we will study the impact the instrumentation has on static analysis tools, notably on their success rate. Additionally, we will study with the analysis of a handcrafted application to check whether the instrumentation does, in fact, improve the results of analysis tools.

5.5.1 Dynamic Analysis Results

After running the dynamic analysis on our dataset the first time, we realised our dynamic setup was quite fragile. We found that 43.09% of the executions failed with various errors. The majority of those errors were related to failures to connect to the Frida agent or start the activity from Frida. Some of those errors seemed to come from Frida, while others seemed related to the emulator failing to start the application. We found that relaunching the analysis for the applications that failed was the simplest way to fix those issues, and after 6 passes, we went from 2136 to 209 applications that could not be analysed. The remaining errors look more related to the application itself or Android, with 96 errors being a failure to install the application, and 110 others being a null pointer exception from Frida.

Unfortunately, although we managed to start the applications, we can see from the list of activities visited by GroddDroid that a majority (84.77%) of the applications stopped before even starting one activity. Some applications do not have any activities and are not intended to interact with a user, but those are clearly a minority and do not explain such a high number. We expected some issues related to the use of an emulator, like the lack of x86_64 library in the applications, or countermesures aborting the application if an emulator is detected. We manually looked at some applications, but did not find a notable pattern. In some cases, the application was just broken – for instance, an application was trying to load a native library that simply does not exist in the application. In other cases, Frida is to blame: we found some cases where calling a method from Frida can confuse the ART. `protected` methods cannot be called from a class other than the one that defined the method or one of its children. The issue is that Frida might be considered by the ART as another class, leading to the ART aborting the application. Table 11 shows the number of applications that we analysed, if we managed to start at least one activity and if we intercepted code loading or reflection. It also shows the average number of activities visited (when at least one activity was started). This average is slightly higher than 1, which seems reasonable: a lot of applications do not need more than one activity, but some do, and we did manage to explore at least some of those additional activities. As shown in the table, even if the application fails to start an activity, sometimes it will still load external code or use reflection.

We later tested the applications on a real phone (model Nothing (2a), Android 15), without Frida but still using GroddRunner. This time, we managed to visit at least one activity for 2130 applications, 3 times more than in our actual experiment. This shows that our setup is indeed breaking applications, but also that there is still another issue we did not find: more than half of the tested applications did not display any activities at all.

The high number of applications that did not start an activity means that our results will be highly biased. The code/method that might be loaded/called by reflection from inside activities

	nb apk	nb failed		activities visited		average nb activities when > 0
		1^{st} pass	6^{th} pass	0	≥ 1	
All	4957	2136	209	4025	723	1.3
With Reflection	3948			3298	650	1.3
With Code Loading	598			453	145	1.2

Table 11: Summary of the dynamic exploration of the applications from the RASTA dataset collected by Androzoo in 2023

Nb Occurences	SHA 256	Content	Format
273	bee390afa2...	Lcom/facebook/ads/*	DEX
98	7aae06433c...	Lcom/facebook/ads/*	DEX
70	920e465a87...	Lcom/facebook/ads/*	DEX
31	51dd5ff34a...	Lcom/google/android/ads/*	APK
12	d44cfb6b41...	Lcom/google/android/ads/*	APK
9	be87bb0a50...	Lcom/facebook/ads/*	DEX
9	26fb1a7903...	Lcom/facebook/ads/*	DEX
7	8395a0121e...	Lcom/facebook/ads/*	DEX
7	1eea5584eb...	Lcom/google/android/ads/*	APK
6	94f66aa1ae...	Lcom/appsflyer/internal/*	DEX
		...	

Table 12: Most common dynamically loaded files

is filtered out by the limit of our dynamic execution. This bias must be kept in mind while reading the next subsection that studies the bytecode that we intercepted.

5.5.2 The Bytecode Loaded by Application

We collected a total of 640 files for 598 application that we detected loading bytecode dynamically. 92 of them were loaded by a `DexClassLoader`, 547 were loaded by a `InMemoryDexClassLoader`, and 1 was loaded by a `PathClassLoader`.

Once we compared the files, we found that we only collected 70 distinct files, and that 273 were identical. Once we looked more in detail, we found that most of those files are advertisement libraries. In total, we collected 87 files containing Google ads libraries and 492 files containing Facebook ads libraries. In addition, we found 48 files containing code that we believe to be AppsFlyer, a company that provides "measurement, analytics, engagement, and fraud protection technologies". The remaining 13 files were custom code from high security applications (*i.e.*, banking, social security). Table 12 summarises the information we collected about the most common bytecode files.

5.5.3 Impact on Analysis Tools

We took the applications associated with the 13 unique DEX files we found to see the impact of our transformation.

The applications were indeed obfuscated, making a manual analysis tedious. We did not find visible DEX files or APK files inside the applications, meaning the applications are either

downloading or generating them from variables and assets at runtime. To estimate the scope of the code we made available, we use Androguard to generate the call graph of the applications, before and after the instrumentation. Table 13 shows the number of edges of those call graphs. The columns before and after show the total number of edges of the graphs, and the diff column indicates the number of new edges detected (*i.e.*, the number of edges after instrumentation minus the number of edges before). This number include edges from the bytecode loaded dynamically, as well as the call added to reflect reflection calls, and calls to "glue" methods (method like `Integer.intValue()` used to convert objects to scalar values, or calls to `T.check_is_Xxx_xxx(Method)` used to check if a `Method` object represents a known method). The last column, "Added Reflection", is the list of non-glue method calls found in the call graph of the instrumented application but neither in the call graph of the original APK, nor in the call graphs of the added bytecode files that we computed separately. This corresponds to the calls we added to represent reflection calls.

The first application, 0019d7fb6a..., is noticable. The instrumented APK has ten times more edges to its call graph than the original, and only one reflection call. This is consistent with the behaviour of a packer: the application loads the main part of its code at runtime and switches from the bootstrap code to the loaded code with a single reflection call.

Unfortunately, our implementation of the transformation is imperfect and sometimes fails, as illustrated by 5d2cd1d10a… in Table 13. However, over the 4748 applications whose dynamic analysis finished in our experiment, 4681 were patched. The remaining 1.41% failed either due to some quirk in the zip format of the APK file, because of a bug in our implementation when exceeding the method reference limit in a single DEX file, or in the case of 5d2cd1d10a…,

APK SHA 256	Number of Call Graph edges			
	Before	After	Diff	Added Reflection
0019d7fb6a…	641	60 170	59 529	1
274b677449…	537 613	540 674	3061	26
34599c2499…	336 740	339 616	2876	29
35065c6834…	343 245	346 694	3449	26
e7b2fb02ff…	464 642	465 389	747	91
efececc03c…	243 647	243 925	278	23
f34ce1e7a8…	704 095	706 576	2481	28
5d2cd1d10a…	*Instrumentation Crashed*			

Table 13: Edges added to the call graphs computed by Androguard by instrumenting the applications

because the application reused the original application classloader to load new code instead of instanciated a new class loader (a behavior we did not expect as possible using only the SDK, but enabled by hidden APIs). Taking into account the failure from both dynamic analysis and the instrumentation process, we have a 5.57% failure rate. This is a reasonable failure rate, but we should keep in mind that it adds up to the failure rate of the other tools we want to use on the patched application.

To check the impact on the finishing rate of our instrumentation, we then run the same experiment we ran in Chapter 3. We ran the tools on the APK before and after instrumentation, and compared the finishing rates in Figure 18 (without taking into account APKs we failed to patch[1]).

The finishing rate comparison is shown in Figure 18. We can see that in most cases, the finishing rate is either the same or slightly lower for the instrumented application. This is consistent with the fact that we add more bytecode to the application, hence adding more opportunities for failure during analysis. They are two notable exceptions: Saaf and IC3. The finishing rate of IC3, which was previously reasonable, dropped to 0 after our instrumentation, while the finishing rate of Saaf jumped to 100%, which is extremely suspicious. Analysing the logs of the analysis showed that both cases have the same origin: the bytecode generated by our instrumentation has a version number of 37 (the version introduced by Android 7.0). Unfortunately, neither the

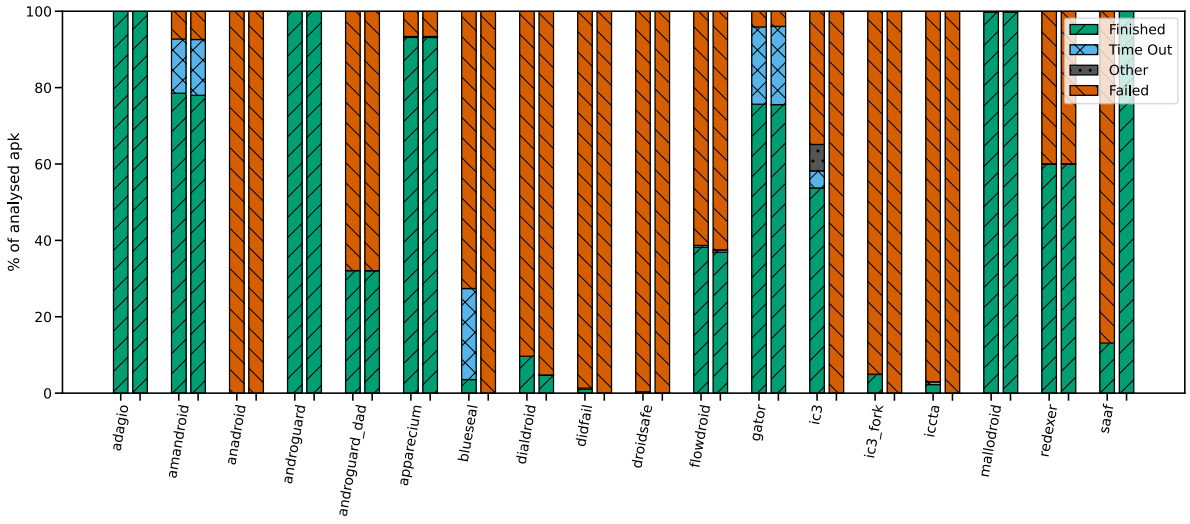


Figure 18: Exit status of static analysis tools on original APKs (left) and patched APKs (right)

1. Due to a handling error during the experiment, the figure shows the results for 4274 APKs instead of 4681. We also ignored the tool from Wognsen *et al.* [67] due to the high number of timeouts

version of Apktool used by Saaf nor Dare (the tool used by IC3 to convert Dalvik bytecode to Java bytecode) recognises this version of bytecode, and thus failed to parse the APK. In the case of Dare and IC3, our experiment correctly identifies this as a crash. On the other hand, Saaf do not detect the issue with Apktool and pursues the analysis with no bytecode to analyse and returns a valid return file, but for an empty application.

5.5.4 Example

In this subsection, we use our approach on a unique APK to look in more detail into the analysis of the transformed application. We handcrafted this application for the purpose of demonstrating how this can help a reverse engineer in their work. Accordingly, this application is quite small and contains both dynamic code loading and reflection. We defined a method `Utils.source()` and `Utils.sink()` to model a method that collects sensitive data and a method that exfiltrates data respectively. Those methods are the ones we will use with Flowdroid to track data flows.

A first analysis of the content of the application shows that the application contains one `Activity` that instantiates the class `Main` and calls `Main.main()`. Listing 13 shows most of the code of `Main` as returned by Jadx. We can see that the class contains another DEX file encoded in base 64 and loaded in the `InMemoryDexClassLoader cl` (line 7). A class is then loaded from this class loader (line 11), and two methods from this class loader are called (line 14). The names of this class and methods are not directly accessible as they have been ciphered and are decoded just before being used at runtime. Here, the encryption key is available statically (line 6), and in theory, a very good static analyser implementing Android `Cipher` API could compute the actual methods called. However, we could easily imagine an application that gets this key from a remote command and control server. In this case, it would be impossible to compute those methods with static analysis alone. When running Flowdroid on this application, it computed a call graph of 43 edges on this application, and no data leaks. This is not particularly surprising considering the obfuscation methods used.

Then we run the dynamic analysis we described in Section 5.4 on the application and apply the transformation described in Section 5.3 to add the dynamic information to it. This time, Flowdroid computes a larger call graph of 76 edges, and does find a data leak. Indeed, when looking at the new application with Jadx, we notice a new class `Malicious`, and the code of `Main.main()` is now as shown in Listing 14: the method called in the loop is either `Malicious.get_data`, `Malicious.send_data()` or `Method.invoke()` (lines 9, 11 and 12). Although self-explanatory, verifying the code of those methods indeed confirms that `get_data()` calls `Utils.source()` and `send_data()` calls `Utils.sink()`.

For a higher-level view of the method, we can also look at its call graph. We used Androguard

```java
package com.example.theseus;

public class Main {
    private static final String DEX = "ZGV4CjA [...] EAAABEAwAA";
    Activity ac;
    private Key key = new SecretKeySpec("_-_Secret Key_-_".getBytes(),
"AES");
    ClassLoader cl = new
InMemoryDexClassLoader(ByteBuffer.wrap(Base64.decode(DEX, 2)),
Main.class.getClassLoader());

    public void main() throws Exception {
        String[] strArr = {"n6WGYJzjDrUvR9cYljlNlw==", "dapES0wl/
iFIPuMnH3fh7g=="};
        Class<?> loadClass =
this.cl.loadClass(decrypt("W5f3xRf3wCSYcYG7ckYGR5xuuESDZ2NcDUzGxsq3sls="));
        Object obj = "imei";
        for (int i = 0; i < 2; i++) {
            obj = loadClass.getMethod(decrypt(strArr[i]),
String.class, Activity.class).invoke(null, obj, this.ac);
        }
    }
    public String decrypt(String str) throws Exception {
        Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
        cipher.init(2, this.key);
        return new String(cipher.doFinal(Base64.decode(str, 2)));
    }

    ...
}
```

Listing 13: Code of the main class of the application, as shown by Jadx, before patching

to generate the call graphs in Figure 19 and Figure 20[1]. Figure 19 shows the original call graph, and gives a good idea of the obfuscation methods used: we can see calls to `Main.decrypt(String)` that itself calls cryptographic APIs, as well as calls to `ClassLoader.loadClass(String)`, `Class.getMethod(String, Class[])` and `Method.invoke(Object, Object[])`. This indicates reflection calls based on ciphered strings, but does not reveal what the method actually does. In comparison, Figure 20, the call graph after instrumentation, still shows the cryptographic and reflection calls, as well as four new method calls. In grey on the figure, we can see the glue methods (`T.check_is_Xxx_xxx(Method)`). Those methods are part of the instrumentation

1. We manually edited the generated .dot files for readability.

```
 1  public void main() throws Exception {
 2      String[] strArr = {"n6WGYJzjDrUvR9cYljlNlw==", "dapES0wl/
iFIPuMnH3fh7g=="};
 3      Class<?> loadClass =
this.cl.loadClass(decrypt("W5f3xRf3wCSYcYG7ckYGR5xuuESDZ2NcDUzGxsq3sls="));
 4      Object obj = "imei";
 5      for (int i = 0; i < 2; i++) {
 6          Method method = loadClass.getMethod(decrypt(strArr[i]),
String.class, Activity.class);
 7          Object[] objArr = {obj, this.ac};
 8          obj =
T.check_is_Malicious_get_data_fe2fa96eab371e46(method) ?
 9              Malicious.get_data((String) objArr[0], (Activity)
objArr[1]) :
10              T.check_is_Malicious_send_data_ca50fd7916476073(method) ?
11              Malicious.send_data((String) objArr[0], (Activity)
objArr[1]) :
12              method.invoke(null, objArr);
13      }
14  }
```

Listing 14: Code of `Main.main()`, as shown by Jadx, after patching

process presented in Section 5.3, but do not bring a lot to the analysis of the call graph. In red on the figure however, we have the calls that were hidded by reflection in the first call graph, and thank to the bytecode of the methods called being injected in the application, we can also

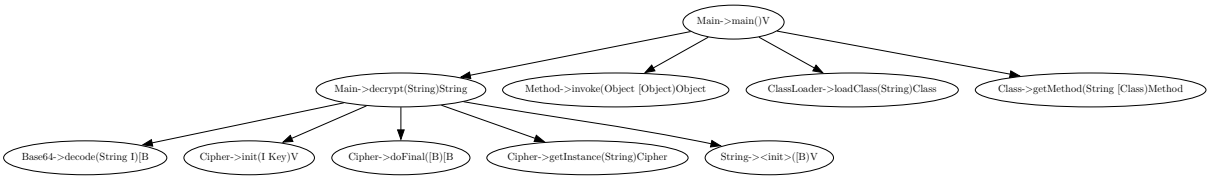


Figure 19: Call Graph of `Main.main()` generated by Androguard before patching

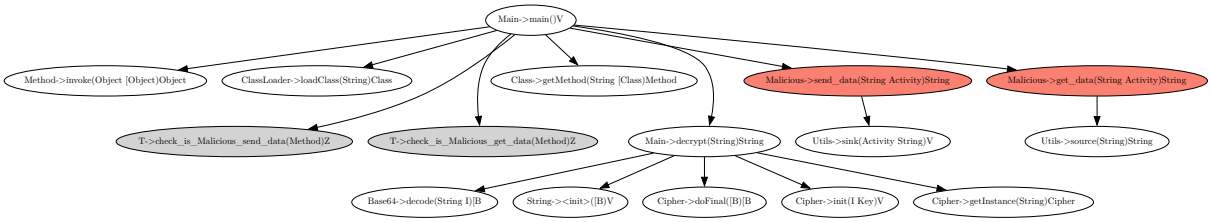


Figure 20: Call Graph of `Main.main()` generated by Androguard after patching

see that they call `Utils.source(String)` and `Utils.sink(String)`, the methods we defined for this application as source of confidential data and exfiltration method.

5.5.5 Androscalpel Performances

Because we implemented our own instrumentation library, we wanted to compare it to other existing options. Unfortunately, we did not have time to compare the robustness and correctness of the generated applications. However, we did compare the performances of our library, Androscalpel, to Apktool and Soot, over the first 100 applications of RASTA (in alphabetical order of the SHA256).

Due to time constraints, we could not test a complex transformation, as adding registers requires complex operations for both Androscalpel and Apktool (see Section 5.3.4 for more details). We decided to test two operations: travelling the instructions of an application (a read-only operation), and regenerating an application, without modification (a read/write operation). It should be noted that all three of the tested tools have multiprocessing support, but we disabled the option when testing the generation of an application with Soot, as it raised errors.

Table 14 compares the resources consumed by each tool for each operation. We can see that for read-only operation, we are 16 times faster than Soot and 8 times faster than Apktool, while keeping a smaller memory footprint. When generating an application, the gap lessens, but we are still almost 8 times faster than Soot. Some of this difference probably comes from implementation choices: Soot and Apktool are implemented in Java, which has a noticeable overhead compared to Rust. However, a noticeable part of this difference can also be explained by the specialised nature of our library; we did not implement all the features Soot has, and we do not parse Android resources like Apktool does. Having better performances does not means that our solution can replace the other in all cases.

Tool		Soot	Apktool	Androscalpel
Read	Time (s)	73.95	33.09	4.38
	Mem (GB)	2.33	1.38	0.58
	Detected Crashes	0	0	0
Read/Write	Time (s)	156.58	74.89	20.34
	Mem (GB)	4.02	1.92	1.19
	Detected Crashes	10	4	1

Table 14: Average time and memory consumption of Soot, Apktool and Androscalpel

Nevertheless, it should be noted that over the 100 applications tested, Soot failed to regenerate 10 of them, Apktool 4, and Androscalpel only 1, showing that our efforts to limit crashes were successful.

∴

To conclude, we showed that our approach indeed improves the results of analysis tools without impacting their finishing rates much. Unfortunately, we also noticed that our dynamic analysis is suboptimal, either due to our experimental setup or due to our solution to explore the applications. In the next section, we will present in more detail the limitations of our solution, as well as future work that can be done to improve the contributions presented in this chapter.

5.6 Limitations and Future Works

The method we presented in this chapter has a number of underdeveloped aspects. In this section, we will present those issues and potential avenues of improvement related to the bytecode transformation, the dynamic analysis and DroidRA, a tool similar to our solution.

5.6.1 Bytecode Transformation

Custom Class Loaders The first obvious limitation of our bytecode transformation is that we do not know what custom class loaders (class loaders implemented by the application developer, as opposed to the class loaders in the SDK) do, so we cannot accurately reproduce statically their behaviour. For instance, we can imagine a class loader that loads all classes whose name starts with an `A` from one DEX file, and all classes whose name starts with a `B` from another. If both DEX files have colliding classes, our implementation will not select the right classes. We elected to fallback to the behaviour of the `BaseDexClassLoader`, which is the highest Android-specific class loader in the inheritance hierarchy, and whose behaviour is shared by all class loaders except `DelegateLastClassLoader`. The current implementation of the ART enforces some restrictions on the class loader's behaviour to optimise the runtime performance by caching classes. This gives us some guarantees that custom class loaders will keep some coherence with the classic class loaders. For instance, a class loaded dynamically must have the same name as the name used in `ClassLoader.loadClass()`. This makes `BaseDexClassLoader` a good approximation for legitimate class loaders. However, an obfuscated application could use the techniques discussed in Section 4.5.2, in which case our model would be entirely wrong.

It would be interesting to explore if some form of static analysis, like symbolic execution, could be used to extract the behaviour of an ad hoc class loader and be used to model the class used appropriately. A more reasonable approach would be to improve the static analysis to intercept each call of `loadClass()` of each class loader, including implicit calls performed by the ART. This would allow us to collect a mapping $(\text{class loader}, \text{class name}) \rightarrow \text{class}$ that can then be used when renaming colliding classes.

Multiple Class Loaders for one `Method.invoke()` Although we managed to handle calls to different methods from one `Method.invoke()` site, we do not handle calling methods from different class loaders with colliding class definitions. The first reason is that it is quite challenging to compare class loaders statically. At runtime, each object has a unique identifier that can be used to compare them over the course of the same execution, but this identifier is reset each time the application starts. This means we cannot use this identifier in an `if` condition to differentiate the class loaders. Ideally, we would combine the hash of the loaded DEX files, the class loader class and parent to make a unique, static identifier, but the DEX files loaded by a class loader cannot be accessed at runtime without accessing the process memory at arbitrary locations. For some class loaders, the string representation returned by `Object.toString()` lists the location of the loaded DEX file on the file system. This is not the case for the commonly used `InMemoryClassLoader`. In addition, the DEX files are often located in the application's private folder, whose name is derived from the hash of the APK itself. Because we modify the application, the path of the private folder also changes, and so will the string representation of the class loaders. Checking the class loader of a class can also have side effects on class loaders that delegate to the main application class loader: because we inject the classes in the APK, the classes of the class loader are now already in the main application class loader, which in most cases will have priority over the other class loaders, and lead to the class being loaded by the application class loader instead of the original class loader. If we check for the class loader, we would need to consider such cases and rename each class of each class loader before reinjecting them in the application. This would greatly increase the risk of breaking the application during its transformation. Instead, we elected to ignore the class loaders when selecting the method to invoke. This leads to potential invalid runtime behaviour, as the first method that matches the class name will be called, but the alternative methods from other class loaders still appear in the new application, albeit in a block that might be flagged as dead code by a sufficiently advanced static analyser.

`ClassNotFoundException` *may not be raised* In the very specific situation where the original application tries to access a class from dynamically loaded bytecode without actually accessing this bytecode (*e.g.*, by using the wrong class loader), the patched application behaviour will differ. The original application should raise a `ClassNotFoundException`, but in the patched application, the class will be accessible, and the exception will not be raised. In practice, there is not a lot of reasons to do such a thing. One could be to check if the APK has been tampered with, but there are easier ways to do this, like checking the application signature. Another would be to check if the class is already available, and if not, load it dynamically, in which case it does not matter, as code loaded dynamically is already present. In any case, because we remove neither the calls to the function that load the classes (like `ClassLoader.loadClass(..)`) nor the `try` / `catch` blocks, static analysis tools that can handle the original behaviour should still be able to access the old behaviour.

5.6.2 Dynamic Analysis

Anti Evasion Our dynamic analysis does not perform any kind of anti-evasive technique. Any application implementing even basic evasion will detect our environment and will probably not load malicious bytecode. Running the dynamic analysis in an appropriate sandbox, such as DroidDungeon [55], should improve the results significantly.

Code Coverage In Section 5.5.1, we saw that our dynamic analysis performed poorly. It may be due to our experimental setup, and it is possible that a better sandbox will fix the issue. However, there is a larger code coverage issue. We tried to manually analyse a few applications marked as malware on MalwareBazaar to test our method. Although we did confirm that the applications were using reflection and dynamic code loading with a static analysis, we did not manage to trigger this behaviour at runtime, and other obfuscation techniques make it very difficult to determine statically the required condition to trigger them. Thus, we believe that techniques to improve code coverage are indeed needed when analysing applications. This could mean better exploration techniques, such as the one implemented by Stoat and GroddDroid, or more intrusive approaches, such as forced execution.

5.6.3 Comparison with DroidRA and Other Tools

It would be very interesting to compare our tool to DroidRA. DroidRA is a tool that computes reflection information using static analysis and patches the application to add those calls. Beyond the classic comparison of static versus dynamic, DroidRA has a similar goal and strategy to ours. Two notable comparison criteria would be the failure rate and the number of edges added to an application call graph. The first criterion indicates how much the results can be used by other tools, while the second indicates how effective the approaches are.

Because we elected to make our own software to modify the bytecode of the APKs, it would be insightful to compare the finishing rate and performances of simple transformations with our tool, to the same transformation made with Apktool, Soot or SootUp (we only compared the performances for re-generating and application without transformations). An example of a transformation to test would be to log each method call and its return value. More than finding which solution is the best to instrument an application, this would allow us to compare the weaknesses of each tool and find if some recurring issues for some tools can be solved using a technical solution implemented by another tool (*e.g.*, some applications deliberately include files with names that crash the standard Java zip library).

5.7 Conclusion

In this chapter, we presented a set of transformations to encode reflection calls and code loaded dynamically inside the application. We also presented a dynamic analysis approach to collect the information needed to perform those transformations.

We then applied this method to a recent subset of applications of our dataset from Chapter 3. When comparing the success rate of the tools of Chapter 3 on the applications before and after the transformation, we found that, in general, the success rate of those tools slightly decreases (except for a few tools). We also showed that our transformation allows static analysis tools to access and process that runtime information in their analysis. However, a more in-depth look at the results of our dynamic analysis showed that our code coverage is lacking, and that the great majority of dynamically loaded code we intercepted is from generic advertisement and telemetry libraries.

Pb3: *Can we use instrumentation to provide dynamic code loading and reflection data collected dynamically to static analysis tools and improve their results?*

We showed that instrumentation can be used to add direct calls to methods initially called through reflections, which, combined with the injection in the application of dynamically loaded bytecode, allows generic static analysis tools to access previously unavailable code. However, we also found that the dynamic analysis can be a significant bottleneck in this approach.

Chapter 6

Conclusion

You know if you would have said so in the beginning, you would have saved yourself a whole lot of trouble.

— Kate "Acid Burn" Libby, Hackers

6.1 Contributions of this Thesis

In this thesis, we presented the following contributions.

First, we explored the reusability of static analysis tools. Based on a systematic literature review by Li *et al.*, we identified 22 tools of interest, published between 2012 and 2017. To estimate the current usability of those tools, we tested their most recent version on a large dataset of 62 525 applications. We then counted the number of analyses that finished and returned a result. We established that 54.55% of 22 tools are not reusable, in particular when the applications are recent. We were not able to use two of them, even with the help of the authors, while 10 others failed to finish their analysis more than half the time. The study of the finishing rate of the tools for applications grouped by their characteristics showed that the greater bytecode size increases the chance of analysis failure. The same goes for min SDK version to a lesser extent, and it appears that analyses of malware are less likely to encounter a fatal error than analyses of goodware. During the testing process, we built Docker images of working setups for the tools. We released those images in the hope of helping future researchers who would want to use those tools.

Our second contribution models the default class loading behaviour of Android and introduces a class of obfuscation based on it: shadow attacks. We showed that, by including multiple classes with the same name in an application, or including classes with the same name as classes in the Android SDK, an application can mislead a reverse engineer or impact the results of analysis tools. We scanned a dataset of recent applications and found that although those situations appear in the wild, shadow attacks do not seem to be actually used. Instead, we believe that

classes from the SDK are added either for retro-compatibility or due to the developer being unaware that a library was already present in the Android SDK, and the few cases where classes are present multiple times in the application appear to be mistakes during the compilation of the application. Still, 23.52% of the applications we tested were shadowing classes from the targeted SDK version.

Lastly, we proposed a solution to reuse any static analysis tool on an application that uses dynamic code loading or reflection. To do so, we collect the relevant information dynamically, then instrument the application to encode the dynamic information inside a valid application mimicking the dynamic behaviour of the original one. This new application can then be analysed normally by any tool that accepts an application as input. We tested our method on a subset of recent applications from the dataset of our first contribution. The results of our dynamic analysis suggest that we failed to correctly explore many applications, hinting at weaknesses in our experimental setup. Nonetheless, we did obtain some dynamic data, allowing us to pursue our experiment. We compared the finishing rate of tools on the original application and the instrumented application using the same experiment as in our first contribution, and found that, in general, the instrumentation only slightly reduces the finishing rate of analysis tools. We also confirmed that the instrumentation improves the result of analysis tools, allowing them to compute more comprehensive call graphs of the applications, or to detect new data flows.

6.2 Perspectives for Future Work

In this section, we present what, in light of this thesis, we believe to be worthwhile avenues of work to improve the Android reverse engineering ecosystem.

The main issues that appeared in all our work appear to be engineering ones. The errors we analysed in Chapter 3 showed that even something that should be basic, reading the content of an application, can be challenging. Chapter 4 also showed that reproducing the exact behaviour of Android is more difficult than it seems (in our specific case, it was the class loading algorithm, but we can expect other features to have similar edge cases). As long as those issues are not solved, we cannot build robust analysis tools.

One avenue that is more research-oriented and that should be investigated would be to reuse for analysis purposes the code actually used by Android. For instance, the parsing of DEX, APK, and resource files could be done using the same code as the ART. This is possible thanks to AOSP being open-source. However, this is not straightforward. Dynamic analysis relying on patched versions of the AOSP showed that it is difficult to maintain this kind of software over time. Doing this would require limiting the modifications to the actual source code of Android to minimise the changes needed at each Android update. Another obstacle to overcome is to decouple the compilation of the tool from the rest of AOSP: it is a massive dependency that needs a lot of resources to build. Having such a dependency would be a barrier to entry,

preventing others from modifying or improving the tool. Should those issues be solved, directly using the code from AOSP would allow such a tool to stay up to date with Android and limit discrepancies between what Android does and what the tool sees.

An orthogonal solution to this problem of not being able to analyse edge cases is to create a new benchmark to test the capacity of a tool to handle real-life applications. Benchmarks are usually targeted at some specific technique (*e.g.*, taint tracking), and accordingly, test for issues specific to the targeted technique (*e.g.*, accurately tracking data that passes through an array). We suggest using a similar method to what we did in Chapter 3 to keep the benchmark independent from the tested tools. Instead of checking the correctness of the tools, this benchmark should test if the tool is able to finish its analysis. Applications in this benchmark could either be real-life applications that was proven to be difficult to analyse (for instance, applications that crashed many of the tested tools in Chapter 3), or hand-crafted applications reproducing corner cases or anti-reverse techniques encountered while analysing obfuscated applications (for instance, an application with gibberish binary file names inside `META-INF/` that can crash Jadx zip reader). The main challenge with such a benchmark is that it would need frequent updates to follow Android evolutions, and be diverse enough to encompass a large spectrum of possible issues.

Lastly, our experience with dynamic analysis led us to believe that there is a need for a new protocol and/or API for automatic testing. Currently, except for intrusive methods like instrumentations, interacting with an application is done through a mix of ADB and UI Automator. Unfortunately, those tools give very poor feedback. Information about the execution is mostly found in the Android logs, lost among other system events, and it is difficult to filter the events related to the application without losing critical data. For instance, exception logs are usually linked to the application in the Android logs, but some exceptions originating from the ART (and not the code of the application itself) are logged as system errors. Similarly, an application can fail to install, or successfully install but later be quarantined or uninstalled by the Android operating system, without clear feedback. Another issue is that more and more, Android will require interactions with the system (for instance, with a security permission pop-up), and those interactions are not handled by UI Automator. Similarly, when an application opens another one for a specific task (opening a document, for example), what is happening is unclear from the standpoint of UI Atomator. We think that an API or protocol that merges and delivers in a structured way all those informations, and allows access to the relevant component (such as a system popup) would be beneficial both for automated testing and dynamic analysis.

Integrating such a protocol into Android would open interesting perspectives. For instance, we could imagine Google requiring applications requesting critical permissions to provide test inputs with a high code coverage (maybe even 100% of coverage). Those tests would incentivise application developers to provide better quality code for applications handling sensitive data, but also to provide solutions for the coverage issue that comes with dynamic analysis. Requiring

a high code coverage would force the developer to supply solutions for situations normally requiring human interaction. For example, if an application requires the user to authenticate themself, the developer would need to provide a testing account that can then be used for tests and analysis. Of course, we can expect malicious applications to implement evasion techniques when they detect an analysis following the tests they provided, but code coverage can be checked, and imposing constraints on the coverage of the tests should mitigate evasion.

Appendices

Appendix A

Released Software and Artifacts

In Chapter 3, we mentioned that we had some difficulties finding some software listed by Li *et al.* following the disappearance of the original websites hosting it. To limit the risk of having the same issue, we hosted the different pieces of software we released for this thesis in several locations. This appendix lists the software we released as well as the different places they can be found.

A.1 RASTA

The code used in Chapter 3 is available at those locations:

- https://gitlab.inria.fr/pirat-public/android/rasta
- https://git.mineau.eu/these-android-re/rasta
- https://github.com/histausse/rasta
- https://doi.org/10.5281/zenodo.10137904

The exact version of the code used in Chapter 3 is tagged as `icsr2024` in the git repositories and corresponds to the one stored in Zenodo.

The results of our experiment and the list of applications in the dataset are also available in the Zenodo archive[1].

The container images used to run the different tools are available on Zenodo at https://doi.org/10.5281/zenodo.10980349 as Singularity images, and on Dockerhub under the names:

- `histausse/rasta-adagio:icsr2024`
- `histausse/rasta-amandroid:icsr2024`
- `histausse/rasta-anadroid:icsr2024`
- `histausse/rasta-androguard-dad:icsr2024`
- `histausse/rasta-androguard:icsr2024`
- `histausse/rasta-apparecium:icsr2024`
- `histausse/rasta-blueseal:icsr2024`
- `histausse/rasta-dialdroid:icsr2024`
- `histausse/rasta-didfail:icsr2024`
- `histausse/rasta-droidsafe:icsr2024`

1. https://doi.org/10.5281/zenodo.10137904

- `histausse/rasta-flowdroid:icsr2024`
- `histausse/rasta-gator:icsr2024`
- `histausse/rasta-ic3-fork:icsr2024`
- `histausse/rasta-ic3:icsr2024`
- `histausse/rasta-iccta:icsr2024`
- `histausse/rasta-mallodroid:icsr2024`
- `histausse/rasta-redexer:icsr2024`
- `histausse/rasta-saaf:icsr2024`
- `histausse/rasta-wognsen:icsr2024`

A.2 Shadow Attack Survey Dataset

The list of applications we scanned in Chapter 4, as well as the lists of platform classes, fields and, methods we extracted from the emulators for Android SDKs 32, 33, and 34, are stored on Zenodo at https://doi.org/10.5281/zenodo.15846481.

The experiment we used to survey in-the-wild applications is available here:

- https://gitlab.inria.fr/pirat-public/android/android_class_shadowing_scanner
- https://git.mineau.eu/these-android-re/android_class_shadowing_scanner
- https://github.com/histausse/android_class_shadowing_scanner

A.3 Theseus

The scripts we used for dynamic analysis and the code implementing the transformations described in Chapter 5 are available at the following locations under GPL license:

- https://gitlab.inria.fr/pirat-public/android/android-of-theseus
- https://git.mineau.eu/these-android-re/android_of_theseus
- https://github.com/histausse/android_of_theseus

The application transformations rely on Androscalpel, the crate we developed to manipulate Dalvik bytecode. Androscalpel can be found at the following locations:

- https://gitlab.inria.fr/pirat-public/android/androscalpel
- https://git.mineau.eu/these-android-re/androscalpel
- https://github.com/histausse/androscalpel

The dataset, results of the dynamic analysis and results of benchmark before and after the instrumentations with Theseus are available on Zenodo at https://doi.org/10.5281/zenodo.17350991.

Appendix B

Résumé Substantiel en Français

Il y a deux réponses à cette question, comme à toutes les questions : celle du savant et celle du poète.

— Ellana Caldin, Le Pacte des Marchombres, Tome 1: Ellana, de Pierre Bottero

---

B.1 Introduction

Android est le système d'exploitation pour téléphones portables le plus utilisé depuis 2014, et depuis 2017, il surpasse même Windows toutes platformes confondues. Cette popularité en fait une cible de choix pour les acteurs malveillants. Il est donc important d'être capable d'analyser une application pour savoir exactement ce qu'elle fait. Ce processus est appelé l'ingénierie inverse.

Beaucoup de travail a été fait dans ce domaine pour les programmes d'ordinateur. Toutefois, les applications Android présentent leur propres difficultées. Par exemple, les applications sont distribuées dans leur format spécifique, le format APK, et le code des applications est lui-même compilé dans un format de code à octets spécifique à Android: Dalvik. La première difficulté pour l'ingénieur·e inverse est donc d'avoir des outils qui comprennent les formats utilisés par Android. Dans le processus d'analyse, une première étape serait alors de lire le contenu de l'application. Des outils comme Apktool peuvent être utilisés pour convertir les fichiers binaire de l'application dans une version lisible par un·e humain·e. D'autres comme Jadx essaient de générer le code source Java depuis le code à octets. Toutefois, les applications Android peuvent être très grosses et il n'est pas toujours possible de les analyser manuellement. D'autres outils ont été développés pour extraire une représentation plus haut niveau du contenu de l'application. Par exemple, Flowdroid a pour objectif de détecter les fuites d'informations: l'utilisateur·ice définit une liste de méthodes qui génèrent des informations sensbiles, et une liste de méthodes qui exfiltrent des informations vers l'extérieur. Flowdroid va alors calculer s'il existe des chemins

dans l'application permettant de relier des méthodes de la première catégorie avec des méthodes de la seconde.

Malheureusement, ces outils sont difficiles à utiliser, et même s'ils fonctionnent sur des applications simples construites dans le but de tester les outils, il n'est pas rare que ces outils échouent sur de vraies applications. Cela pose la problématique suivante: *À quel point les outils d'analyse Android préalablement publiés sont-ils utilisables aujourd'hui, et quels facteurs impactent leur réutilisabilité?*

Il y a deux familles d'analyse: l'analyse statique et l'analyse dynamique. L'analyse statique analyse l'application sans la lancer, alors que l'analyse dynamique examine le comportement de l'application pendant son exécution. Chacune a ses forces et ses faiblesses, et certains problèmes d'analyse sont traditionnellement associés à l'une ou l'autre pour les résoudre. L'un de ces problème est le chargement dynamique de code. Les applications Android sont initiallement prévues pour être codées en Java, Android a donc a hérité de nombreuses fonctionnalités de Java. En l'occurence, Android a un système de chargeur de classes similaire à celui de Java, qui peut être utilisé pour charger, en cour d'exécution, du code extérieur à l'application. Etant donné que ce code chargé dynamiquement n'est pas nécessairement disponible dans l'application initialement, ce problème est relégué à l'analyse dynamique. Toutefois, il semblerait qu'une généralisation hâtive soit souvent faite, et que le système de chargement de classe dans son ensemble soit relégué à l'analyse dynamique. L'absence d'étude détaillée de ce mecanisme nous amène à notre seconde problématique: *Quel est l'algorithme de chargement de classe utilisé par défaut par Android, et est-ce qu'il impacte l'analyse statique?*

Un autre problème usuellement associé à l'analyse dynamique est la réflexion. Android permet à une application de manipuler les classes et méthodes sous forme d'objet. En utilisant cette fonctionnalité, il est donc possible d'appeler une méthode en utilisant son nom sous forme de chaîne de caractère au lieu d'utiliser une instruction Dalvik avec une référence vers la méthode appelée. Ce cas est déjà compliqué à analyser statiquement quand la chaîne de caractère est lisible dans l'application, mais il devient imposible quand elle ne l'est pas (*ex.* la chaîne est envoyée par un serveur externe lors de l'exécution, ou elle est stockée chiffrée et n'est déchiffrée qu'au dernier moment). L'analyse dynamique permet de capturer à la fois le code chargé dynamiquement et les méthodes appelées par réflexion. Toutefois, obtenir ces instructions est insuffisant. Il n'existe pas de solution standard pour transmettre ces données aux outils d'analyse statique, qui pourtant peuvent en avoir besoin pour analyser l'application dans son entièreté. Certaines contributions d'ingénierie inverse ont déjà proposé d'instrumenter (modifier) l'application pour y ajouter les résultats de leur analyse avant de l'analyser avec d'autres outils. Cette approche prometteuse motive notre troisième problématique: *Peut-on utiliser l'instrumentation pour fournir le code chargé dynamiquement et les informations de réflexion collectées dynamiquement aux outils d'analyse statique pour améliorer leurs résultats?*

B.2 Evaluation de la réutilisabilité des outils d'analyse statique pour Android

Dans ce chapitre, nous étudions la réutilisabilité d'outils d'analyse statique publiés entre 2011 et 2017. Le but de cette étude n'est pas de quantifier la précision des outils, car ces outils ont des objectifs finaux différents. Au contraire, dans ce chapitre nous allons considérer comme correct tout résultat renvoyé par les outils, et uniquement compter les occurences où les outils échouent à calculer un résultat quel qu'il soit.

Les questions auxquelles nous voulons répondre sont:

QR1 Quels outils d'analyse statique pour Android vieux de plus de 5 ans peuvent encore être utilisés aujourd'hui avec un effort raisonnable?

QR2 Comment la réutilisabilité des outils évolue-t-elle avec le temps, en particulier pour l'analyse d'applications publiées avec plus de 5 ans d'écart avec l'outil?

QR3 Est-ce que la réutilisabilité des outils change quand on analyse une application bénigne comparé à un maliciel?

Nous basons notre étude sur revue de litérature systématique de Li *et al.* qui liste les contributions accompagnées d'outils sous licence libre. Nous avons retrouvé les outils en questions, listé dans le Table 15. Nous avons éliminé les outils utilisant de l'analyse dynamique en plus de l'analyse statique, et vérifié la présence des sources, de la documentation, et d'un optionnel exécutable si jamais les sources ne peuvent pas être compilées.

Nous avons ensuite sélectionné la version des outils à utiliser. Certains outils ont évolué depuis leur publication, soit en étant maintenus par leurs auteurs, soit suite à un branchement par un autre développeur. Nous avons décidé d'utiliser la dernière version stable en date de 2023 (date de l'étude). Le seul cas de branchement intéressant que nous avons trouvé est celui d'IC3, que nous avons décidé d'inclure en plus de la version originale. Le Table 16 résume cette étape.

Nous avons ensuite exécuté ces outils sur deux jeux d'applications: Drebin, un jeu de maliciels connus pour être vieux et biaisé, et RASTA, un jeu que nous avons échantilloné nous-mêmes pour réprésenter l'évolution des caractéristiques des applications entre 2010 et 2023, d'un total de 62 525 APKs.

Après avoir lancé les outils, nous avons collecté les différents résultats et traces d'exécution. Figure 21 et Figure 22 montrent les résultas des analyses sur les applications de Drebin et RASTA. L'analyse est considérée comme réussie (vert) si un résultat est obtenu, sinon elle a échoué (rouge). Quand l'analyse met plus d'une heure à finir, elle est avortée (bleue). On peut voir que les outils ont d'assez bon résultats sur Drebin, avec 11 outils qui ont un taux de terminaison au dessus de 85%. Sur RASTA par contre, 12 outils (54.55%) ont un taux de terminaison en dessous de 50%. Les outils qui avaient des difficultés avec Drebin ont aussi de

Outil	Disponibilité Bin	Src	Doc	Dépo type	Décision	Commentaires
A3E	–	✓	✓	github	✗	Outil hybride (statique et dynamique)
A5	–	✓	✗	github	✗	Outil hybride (statique et dynamique)
Adagio	–	✓	✓	github	✓	
Amandroid	✓	✓	✓	github	✓	
Anadroid	✗	✓	✓	github	✓	
Androguard	–	✓	✓✓	github	✓	
Android-app-analysis	✗	✓	✓✓	google	✗	Outil hybride (statique et dynamique)
Apparecium	✓	✓	✗	github	✓	
BlueSeal	✗	✓	○	github	✓	
Choi *et al.*	✗	✓	○	github	✗	Nécessite les fichiers sources
DIALDroid	✓	✓	✓	github	✓	
DidFail	✓	✓	○	bitbucket	✓	
DroidSafe	✗	✓	✓	github	✓	
Flowdroid	✓	✓	✓✓	github	✓	
Gator	✗	✓	✓✓	edu	✓	
IC3	✓	✓	○	github	✓	
IccTA	✓	✓	✓	github	✓	
Lotrack	✗	✓	✗	github	○	Auteurs reconnaissent documentation insuffisante.
MalloDroid	–	✓	✓	github	✓	
PerfChecker	✗	✗	○	request	✓	Binaire obtenu des auteurs
Poeplau *et al.*	✗	○	✗	github	✗	Dédié à la sécurisation d'Android
Redexer	✗	✓	✓	github	✓	
SAAF	✓	✓	✓	github	✓	
StaDynA	✗	✓	✓	request	✗	Outil hybride (statique et dynamique)
Thresher	✗	✓	✓	github	○	Ne compile pas même avec l'aide des auteurs
Wognsen *et al.*	–	✓	✗	bitbucket	✓	

binaires, sources: –: non pertinent, ✓: disponible, ○: partiellement disponible, ✗: non fourni
documentation: ✓✓: excellente, MWE, ✓: quelques incohérences, ○: mauvaise qualité, ✗: non disponible
décision: ✓: considéré; ○: considéré mais pas compilé; ✗: sort du cadre de l'étude

Table 15: Outils considérés: disponibilité et réutilisabilité

mauvais résultats sur RASTA, mais d'autres outils avec des résultats acceptables sur Drebin voient leur résultats chuter avec RASTA.

Ces résultats nous permettent de répondre à notre première question **QR1**:

Sur un jeu d'applications réscentes nous considérons que 54.55% des outils sont utilisables. De plus pour les outils que nous avons pu lancer, 54.9% des analyses ont bien terminé correctement.

Outil	**Original** Etoiles	Vivant	**Branchement Vivant** Nb	Utilisable	**Date Dernière Modification**	**Auteurs Contactés**	**Environment** Langage – SE
Adagio	74	✓	0	✗	2022-11-17	✓	Python – U20.04
Amandroid	161	✗	2	✗	2021-11-10	✓	Scala – U22.04
Anadroid	10	✗	0	✗	2014-06-18	✗	Scala/Java/Python – U22.04
Androguard	4430	✓	3	✗	2023-02-01	✗	Python – Python 3.11 slim
Apparecium	0	✗	1	✗	2014-11-07	✗	Python – U22.04
BlueSeal	0	✗	0	✗	2018-07-04	✓	Java – U14.04
DIALDroid	16	✗	1	✗	2018-04-17	✗	Java – U18.04
DidFail	4	✗			2015-06-17	✓	Java/Python – U12.04
DroidSafe	92	✗	3	✗	2017-04-17	✓	Java/Python – U14.04
Flowdroid	868	✓	1	✗	2023-05-07	✓	Java – U22.04
Gator					2019-09-09	✓	Java/Python – U22.04
IC3	32	✗	3	✓	2022-12-06	✗	Java – U12.04 / 22.04
IccTA	83	✗	0	✗	2016-02-21	✓	Java – U22.04
Lotrack	5	✗	2	✗	2017-05-11	✓	Java – ?
MalloDroid	64	✗	10	✗	2013-12-30	✗	Python – U16.04
PerfChecker		✗			–	✓	Java – U14.04
Redexer	153	✗	0	✗	2021-05-20	✓	Ocaml/Ruby – U22.04
SAAF	35	✗	5	✗	2015-09-01	✓	Java – U14.04
Thresher	31	✗	1	✗	2014-10-25	✓	Java – U14.04
Wognsen *et al.*				✗	2022-06-27	✗	Python/Prolog – U22.04

✓: oui, ✗: non, UX.04: Ubuntu X.04

Table 16: Outils selectionnés, branchements, versions selectionnées et environnements d'exécution

Nous avons ensuite étudié l'évolution du taux de terminaison des outils au cours des ans. La Figure 23 montre cette évolution pour les outils codés en Java. On peut noter une tendance générale où le taux de terminaison diminue avec le temps.

Plusieurs facteurs peuvent être responsables. Par exemple, la librairie standard d'Android et le format des applications ont évolué avec les versions d'Android. Un autre changement notable est la taille du code à octets des applications. Les applications les plus récentes ont notablement plus de code.

Pour déterminer le facteur qui influence le plus le taux de terminaison, nous avons étudié des sous-ensembles de RASTA avec certains de ces paramètres fixés. Par exemple, nous avons tracé

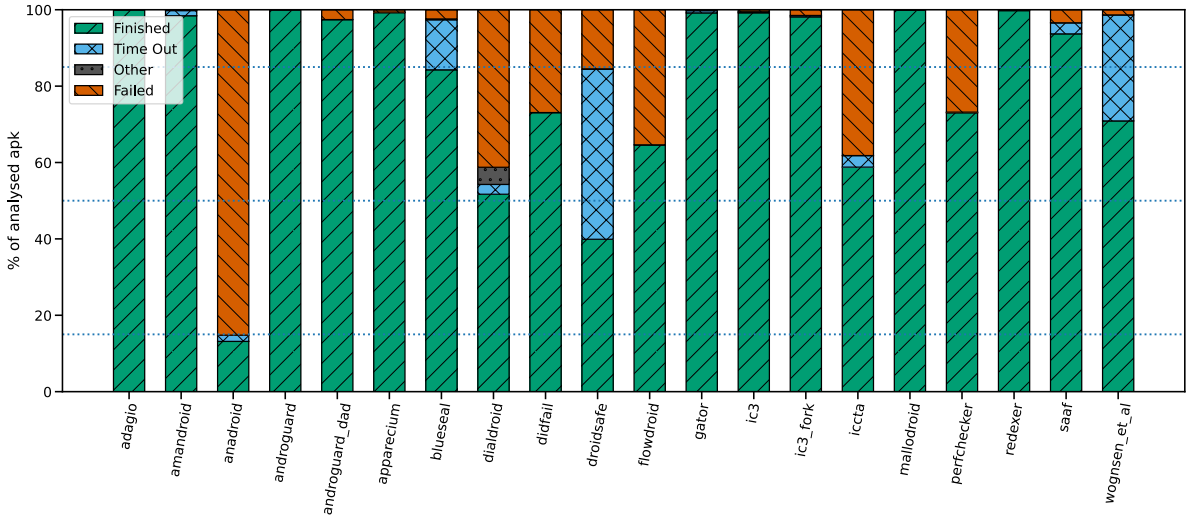


Figure 21: Taux de terminaison pour le jeu d'applications Drebin

l'évolution du taux de terminaison en fonction de l'année de publication des applications sur l'ensemble des applications dont le code à octets fait entre 4.08 et 5.2 Mo.

Nous en avons conlu la réponse à notre question de recherche **QR2**: Au cour du temps, le taux de terminaison des outils diminue, allant de 78% à 61% cinq ans plus tard, à 45% dix ans plus tard. Ce taux varie en fonction de la taille du code à octet, et, dans de moindre mesure, la version d'Android.

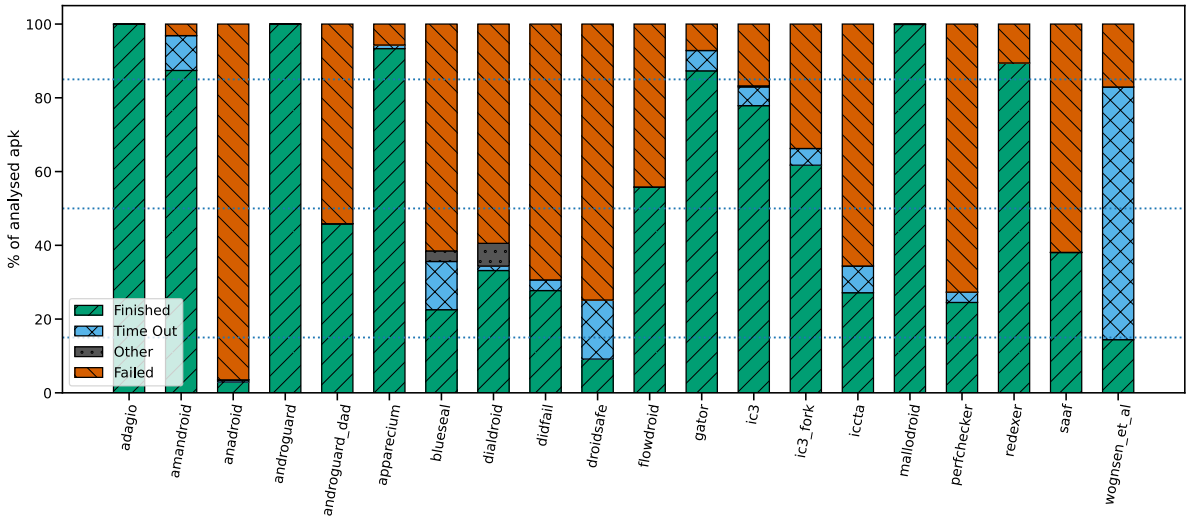


Figure 22: Taux de terminaison pour le jeu d'application RASTA

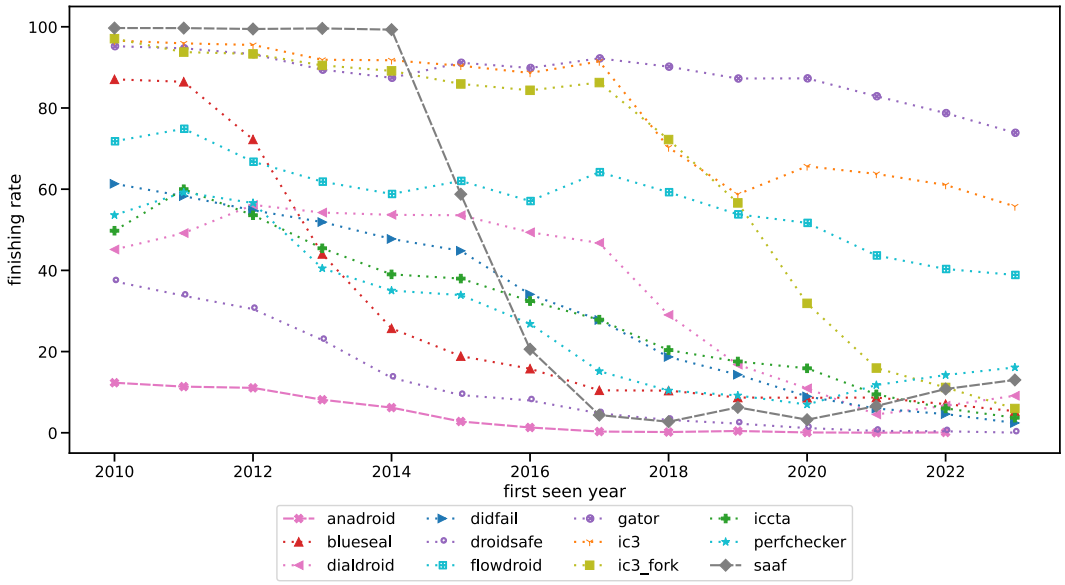


Figure 23: Taux de terminaison des outils basé sur Java au cours des ans

Pour répondre à notre dernière recherche question, nous avons comparé le taux de terminaison entre les applications bénignes et les maliciels. Les résultats semblent indiquer que les maliciels sont plus facilement analysés. Pour vérifier cette affirmation, nous avons comparé le taux de terminaison mais aussi la taille du code à octets des applications et effectivement, il semblerait que ce résultat soit vrai, y compris à taille égale.

Nous avons donc une réponse à notre **RQ3**: Les maliciels causent moins d'erreurs lors de leur analyse par des outils d'analyse statique.

Finalement, nous avons une réponse à notre première problématique:

Plus de la moitié des outils sélectionnés ne sont plus utilisables. Dans certains cas, cela est dû à notre incapacité à les installer correctement, mais majoritairement, cela est dû au faible taux de terminaison des outils lors de l'analyse des applications. Nos résultats montrent que les applications avec beaucoup de code sont plus difficiles à analyser, et, en moindre mesure, la version d'Android ainsi que la malignité de l'application peut avoir un impact.

B.3 Chargeurs de classes au milieu: Dérouter les analyseur statiques pour Android

Dans ce chapitre, nous étudions comment Android gère le chargement de classe en présence de multiples versions de la même classe. Nous modélisons l'algorithme de chargement de classe d'Android, et l'utilisons comme base pour une nouvelle famille de brouillage de code que nous

appelons *masquage de classes*. Nous analysons ensuite des applications publiés en 2023 pour déterminer si cette technique de brouillage est actuellement utilisée.

Le chargement de classe est une fonctionnalité de Java dont Android a hérité. Les développeurs intéragissent avec elle le plus souvent au travers de classes héritant de `ClassLoader` pour charger dynamiquement du code. Toutefois, elle a un rôle bien plus général. À chaque fois qu'Android rencontre une nouvelle classe en exécutant une méthode, il va charger cette classe au travers du mécanisme de chargement de classe.

L'interet de ce mécanisme est qu'il permet d'utiliser des classes provenant de différentes sources. Pour cela, Android associe à chaque classe un objet `ClassLoader`, celui qui a été utilisé pour charger cette classe. Par la suite, Android utilise ce `ClassLoader` pour charger toutes les classes référencées par cette première classe. Pour permettre aux classes provenant de différents `ClassLoader` d'intéragir entre elles, les `ClassLoader` implémentent un mécanisme de délégation. Chaque `ClassLoader` a un "parent", un autre objet de type `ClassLoader`, auquel le `ClassLoader` va déléguer le chargement de classe. Si la classe n'est pas trouvée par le parent, alors le `ClassLoader` va la charger lui même. Bien que ce système de délégation est utilisé par toutes les classes héritant de `ClassLoader` dans la librairie standard d'Android (à l'exception de `DelegateLastClassLoader` qui délègue dans un ordre légèrement différent), ce comportement est spécifié par l'implémentation de chaque classe `ClassLoader`. Une application peut très bien définir une nouvelle classe héritant de `ClassLoader` qui n'implémente pas ce processus. Toutefois, ce cas relève de l'analyse dynamique: un `ClassLoader` défini dans l'application ne peut pas être utilisé par Android sans exécuter du code de l'application pour l'instancier. Dans ce chapitre, nous nous concentrons sur le comportement par défaut d'Android, aussi nous n'avons besoin d'analyser que les `ClassLoader` instanciés par Android lui même pour lancer l'application.

Le premier `ClassLoader` utilisé par Android est `BootClassLoader`. Cette classe est une classe singleton, ce qui signifie qu'il ne peut y avoir qu'une seule instance de la classe par application. Elle est utilisée pour charger les classes de plateforme. Ces classes sont les classes implémentées par Android et qui peuvent être utilisées par une application sans qu'elles ne soient présentes dans l'application. Elles peuvent être séparées en deux catégories, les classes du KDL (Kit de Développement Logiciel) Android, et les classes de l'IPA (Interface de Programmation d'Application) cachée. Les premières forment la librairie standard d'Android. Elles sont documentées et couramment utilisées par les développeurs. Les secondes sont des classes utilisées par Android en interne, mais que les applications ne sont pas supposées utiliser. Elles ne sont pas documentées, et depuis quelques années Android commence à faire des efforts pour empêcher les dévelopeurs de les utiliser. Elles sont toutefois encore utilisées, et, au moins jusqu'à présent, les mesures d'Android ne suffisent pas à les rendre inaccessibles.

Ce `BootClassLoader` est utilisé comme le parent par défaut par tous les `ClassLoader` définis dans les classes plateforme d'Android. Quand le parent d'un `ClassLoader` n'est pas défini (quand sa valeur est nulle), les `ClassLoader` vont déléguer le chargement au `BootClassLoader` à la place. L'autre type de `ClassLoader` utilisé par Android par défaut est le `PathClassLoader`. Cette classe est utilisée pour charger des classes stockées dans des fichiers. Android en définit deux par défaut, un `PathClassLoader` "système", et un `PathClassLoader` pour l'application. La documentation indique que le chargeur "système" est le chargeur par défaut pour le processus principal. Toutefois, il ne semble pas être utilisé en pratique. Le chargeur de l'application en revanche est utilisé pour les classes contenues dans l'application, c'est donc le chargeur utilisé par défaut pour toutes les classes codées par le développeur.

En plus des chargeurs de classes, il y a un dernier critère à considérer. Les fichiers DEX contenant le code à octets des applications ont une limite du nombre de méthodes qui peuvent être référencées. Pour y remédier, Android a introduit un nouveau format d'application contenant plusieurs fichiers DEX. Pour notre étude, le point notable de ces appplications est que bien qu'Android teste qu'un fichier DEX ne contient qu'une seule implémentation de chaque classe, ce test n'est fait que fichier par fichier: deux fichiers DEX peuvent contenir une implémentation d'une même classe chacun. Les fichiers DEX de ces applications "multi-dex" sont nommés `classes.dex`, puis `classesX.dex` où `X` est un entier supérieur ou égale à 2. Pour savoir quelle implémentation est utilisée par Android, il faut donc savoir dans quel ordre les fichiers sont visités par les `PathClassLoader`.

Finalement, après avoir étudié le code source d'Android, nous concluons que l'algorithme utilisé est le même que celui que nous avons décrit dans le pseudo code Code 15. Cet algorithme a deux points notables. En premier lieu, les classes platformes ont toujours la priorité sur les autres classes. Cela peut être intuité pour les classes courantes comme `String`, mais il faut se rappeler que les classes de l'IPA cachée ne sont pas documentées. Ensuite, les classes sont sélectionnées parmis les fichiers DEX dans un ordre non trivial, et s'arrête à la première implémentation trouvée. Le premier fichier testé est `classes.dex`, suivi de `classes2.dex`, puis `classes3.dex` et ainsi de suite, jusqu'à ce qu'un fichier `classesX.dex` n'existe pas. La limite au nombre de fichiers DEX est très élevée (2^{64} sur les téléphones actuels), tant que le fichier suivant existe et que la classe n'est pas trouvée, Android va continuer. Aussi, le code contenu fichier `classes100.dex` peut être utilisé par Android, ou non, par exemple si `classes99.dex` n'existe pas. Plus surprenant, le code contenu dans un fichier `classes1.dex` ou `classes02.dex` ne sera pas utilisé. Lors de l'analyse statique d'applications, ces deux points peuvent mener à des complications que nous allons maintenant explorer.

A partir de cet algorithme, nous avons mis au point plusieurs méthodes de brouillage de code que nous appelons *masquage de classe*: la classe utilisée est masquée par une autre implémentation fournie par le développeur. Nous nous concentrons sur l'obfuscation statique, mais cette stratégie

```
def obtenir_multi_dex_classes_nom_dex(indice: int):
  if indice == 0:
    return "classes.dex"
  else:
    return f"classes{indice+1}.dex"

def charge_classe(nom_classe: str):
  if est_class_platforme(nom_classe):
    return charge_depuis_chargeur_class_boot(nom_classe)
  else:
    indice = 0
    fichier_dex = obtenir_multi_dex_classes_nom_dex(indice)
  while fichier_existe_dans_apk(fichier_dex) and \
    not classe_non_trouvee_dans_fichier_dex(nom_classe, fichier_dex):
    indice += 1
  if fichier_existe_dans_apk(fichier_dex):
    return charge_depuis_fichier(fichier_dex, nom_classe)
  else:
    raise ErreurClasseNonTrouvee()
```

Code 15: Algorithme de chargement de classe par défaut pour les applications Android

peut être étendue à une approche dynamiquement en utilisant différents chargeurs de classes. Nous proposons trois techniques dans cette catégorie:

Auto masquage Ici, le développeur utilise le format multi-dex pour mettre plusieurs implémentations différentes dans la même application. L'objectif est d'exploiter les divergences entre l'algorithme de chargement de classes d'Android et la façon dont les outils d'analyse selectionnent l'implémentation à utiliser. De cette façon, la classe utilisée par Android ne sera pas celle analysée.

Masquage de KDL Cette fois, le développeur inclut une implémentation pour une classe du KDL dans l'application. Un outil qui ne priorise pas les classes plateformes, ou ne les connaît pas, va alors utiliser une implémentation invalide de la classe pour son analyse.

Masquage d'IPA Cachée L'idée est la même que pour la technique précédente, mais cette fois pour une classe de l'IPA caché. Nous distinguons masquage de KDL et masquage d'IPA caché car les IPA cachés n'étant pas documentés, il est possible que des outils soient capables de résoudre la première technique mais pas la deuxième.

Nous avons vérifié l'effet de ses techniques sur 4 outils de rétro-ingénierie Android courants: Jadx, Apktool, Androguard et Flowdroid. Le Table 17 résume nos conclusions. Jadx est un décompilateur d'application. Lorsqu'il est utilisé pour décompiler une application usant d'auto-masquage, il va sélectionner la mauvaise classe, mais indiquer en commentaire la liste des fichiers

de code à octet contenant une implementation de la classe. Apktool et Androguard listent toutes les classes de l'application, il revient donc à l'analyste de choisir la bonne implémentation, et pour les analyses plus poussées d'Androguard, Androguard choisit la mauvaise classe. Aucun de ces trois outils n'indiquent en aucune façon qu'une classe est déjà définie dans le KDL ou les IPA cachés. Flowdroid en revanche est capable de détecter les flux de données passant par des classes du KDL, y compris en présence d'une réimplémentation dans l'application. Ce n'est par contre pas le cas pour les classes d'IPA cachées. Il est intéressant de noter que Soot, la librairie sur laquelle est basé Flowdroid, a bien un algorithme qui priorise les fichers DEX, et que cet algorithme est très proche de celui d'Android. Toutefois, les fichiers commençant par `classes` sont ensuite priorisés par ordre alphabétique. Cela signifie que les classes contenues dans `classes0.dex`, `classes02.dex` ou `classes10.dex` sont priorisées sur celles de `classes2.dex`. Ce problème est hérité par Flowdroid, ce qui le rend sensible à la technique d'auto-masquage.

Pour savoir si ces techniques sont utilisées dans la nature, nous avons scanné 49 975 applications publiées entre janvier 2023 et 2024. Pour vérifier que les différentes implémentations sont bien distinctes, nous comparons la représentation smali (le langage assembleur pour le format Dalvik) du code à octets des méthodes. La Table 18 résume ces résultats. Il est notable qu'un nombre important d'applications (23.52%) ont au moins un cas de masquage. En étudiant en détail, nous avons noté que la majorité des classes concernées sont des classes introduites entre la version minimale et la version cible d'Android pour l'application. Cela laisse entendre que ces classes ont été rajoutées pour permettre à l'application de fonctionner avec les versions d'Android où ces classes n'existent pas. Le taux élevé de code identique pour les cas d'auto-masquage semble également pointer vers des erreurs lors de la compilation de l'application. De plus, l'analyse

Outil	Version	Masquage		
		Auto	KDL	Cachée
Jadx	1.5.0	○	●	●
Apktool	2.9.3	○	●	●
Androguard	4.1.2	○	●	●
Flowdroid	2.13.0	●	×	●

●: Le masquage fonctionne
○: Le masquage fonctionne, mais un avertissement est émis ou les différentes implémentations sont visibles
×: Le masquage ne marche pas

Table 17: Résultats des techniques de masquage contre des outils d'analyse statique

	Nombre d'app			Classes Masquées	Moyenne Médianne	KDL	Min KDL	Code Identique
		%	% maliciel					
Pour toutes les applications du jeu								
Auto	49 975	100.0%	0.53%	2.1	0	32.1	21.7	74.8%
KDL	49 975	100.0%	0.53%	6.5	0	32.1	21.7	8.04%
Cachées	49 975	100.0%	0.53%	0.5	0	32.1	21.7	17.42%
Totale	49 975	100.0%	0.53%	9	0	32.1	21.7	23.76%
Pour les applications avec au moins 1 cas de masquage								
Auto	234	0.47%	5.98%	438.1	18	31.4	22.4	74.8%
KDL	11 755	23.52%	0.38%	27.6	5	32.4	22	8.04%
Cachées	1556	3.11%	0.71%	16.1	1	32.1	22.2	17.42%
Totale	12 301	24.61%	0.42%	36.7	6	32.4	22	23.76%

Table 18: Classes masquées comparées aux classes platform d'Android IPA 34 pour un jeu de 49 975 applications

manuelle des cas où le smali diffère montre que les différences viennent de détails lors de la compilation (par exemple, l'inversion de deux registres, ce qui n'a aucun effet sur l'exécution du code).

Notre conclusion est que le masquage de classes n'est pas activement exploité pour le brouillage de code. En revanche, cette situation se produit naturellement dans les applications. Il est donc important pour les outils d'analyses de modéliser correctement le processus de chargement de classes. Avec l'algorithme décrit par le Code 15, cela répond à notre seconde problématique.

B.4 L'Application de Thésée: Même après avoir ajouté les informations d'exécution, c'est toujours votre application

Dans ce dernier chapitre, nous nous penchons sur la question de comment permettre aux outils d'analyse statique d'accéder à des résultats dynamiques. Des contributions précédentes ont encodé leur résulats dans l'application elle-même pour transmettre leurs résultats à d'autres outils. Nous allons utiliser cette approche pour permettre à des outils d'analyse statique d'analyser le comportement d'applications utilisant de la réflexion ou du chargement de code dynamique.

Premièrement, il nous faut définir la transformation que nous voulons effectuer. Concernant la réflexion, il y a 3 méthodes permettant d'appeler des méthodes arbitraires dans Android. `Class.newInstance()` et `Constructor.newInstance()` permettent d'instancier un nouvel object et d'appeler l'un des ses constructeurs, tandis que `Method.invoke()` permet d'appeler une méthode. Les objects `Class`, `Constructor` ou `Method` utilisés pour appeler ces méthodes peuvent être obtenus de multiples façons. Nous n'allons donc pas chercher à modifier le code obtenant ces objets. À la place, nous allons nous concentrer sur l'appel des méthodes. À différents moments de

l'exécution, une même instruction appelant `Method.invoke()` peut appeler différentes méthodes. De plus, la collection des informations de réflexion ne sera jamais parfaite: il y a des situations où on ne peut jamais être certain d'avoir la liste complète des méthodes appelées. Par exemple, on peut imaginer une application qui appelle par réflexion une méthode dont le nom est obtenu depuis un serveur distant. Dans ce cas, sans accès au code du serveur il est impossible d'avoir la liste exhaustive des méthodes qui peuvent être utilisées. Pour prendre en compte ces deux cas, nous allons remplacer les appels par des blocs conditionnels. Pour chaque méthode dont on sait qu'elle peut être appelée, nous testons si l'object `Method` correspond à cette méthode en comparant son nom et sa signature. Si c'est le cas, la méthode est appelée dans le bloc avec l'instruction Dalvik appropriée. Si la méthode ne correspond à aucune méthode connue, alors l'appel est fait par réflexion. Ainsi, le comportement de l'application est conservé.

Pour le chargement de code dynamique, nous avons conclu qu'il n'est pas nécessaire de modifier le code. A la place, nous pouvons directement ajouter à l'application le ficher DEX en utilisant le format multi-dex. Toutefois, si jamais certaines classes contenues dans le code que nous injectons sont déja présentes dans l'application, nous renommons la classe et ses références de sorte à reproduire statiquement le comportement de l'algorithme de chargement de classe d'Android. Cette approche a des limites, en particulier lors d'appel réflexifs, car il n'existe pas de solutions pour comparer les chargeurs de classes avec une valeur statique. Si un même site d'appel réflexif appelle deux méthodes avec des signatures et noms identiques mais associées à des classes provenant de chargeurs de classes différents, nous ne sommes pas en mesure de reproduire exactement le même comportement statiquement. Toutefois, les appels aux deux différentes méthodes apparaissent bien dans la nouvelle application, ce qui devrait permettre aux outils d'analyse statique de considérer les deux cas possibles.

Pour pouvoir effectuer ces transformations, il nous faut certaines informations. Les noms, signature et chargeur de classes des méthodes appelées par réflexion, ainsi que la position exacte du site de l'appel réflexif, et le code à octets chargé dynamiquement. Pour obtenir ces informations, nous utilisons Frida, un outil permettant d'injecter des scripts dans les méthodes appelées pendant l'exécution d'une application Android. Pour la réflexion, nous avons bien entendu instrumenté `Class.newInstance()`, `Cconstructor.newInstance()` et `Method.invoke()`. Pour le chargement de code, le choix est un peu moins évident car il existe de multiples façons de charger du code à octets. Nous avons finalement choisi `DexFile.openInMemoryDexFileNative()` et `DexFile.openDexFileNative()`, des méthodes de l'IPA cachée. Ces méthodes sont les dernières méthodes appelées dans l'environement Java avant de passer en native pour analyser et charger le code à octets. Pour aider à l'exploration des applications, nous avons réutilisé une partie de GroddDroid, un outil dédié à l'exploration dynamique d'applications.

Nous avons lancé notre analyse statique sur les 5000 applications publiées en 2023 du jeu d'applications RASTA. Malheureusement, les résultats semblent indiquer que notre environ-

nement d'exécution est insuffisant et que beaucoup d'applications n'ont pas été visitées correctement. Malgré tout, nous avons collecté 640 fichiers de code à octets. Toutefois, une fois comparé, nous remarquons que parmi ces fichiers, il n'y a que 70 fichiers distincts. L'inspection du contenu montre que ces fichiers sont principalement des librairies de code publicitaire ou analytiques. Seuls 13 fichiers parmi les 640 collectés ne proviennent ni de Google, ni de Facebook, ni de AppsFlyer. Ces fichiers restants contiennent du code spécifique aux applications les utilisant, principalement des applications exigeant un niveau important de sécurité comme des applications bancaires ou d'assurance santé.

La Table 19 montre le nombre d'arcs du graphe d'appel de fonction de ces quelques applications qui chargent du code spécifique à leur usage dynamiquement. La colonne "Réflexion Ajoutées" correspond au nombre d'appels réflexifs ajoutés à l'application. Les autres arcs ajoutés sont soit des fonctions "colle" que nous avons ajoutées à l'application pour choisir la bonne méthode à appeler reflexivement, soit des méthodes appelées par du code chargé dynamiquement auquel Androguard n'avait pas accès avant l'instrumentation. On peut voir que notre méthode permet à Androguard de calculer un plus grand graphe.

Nous avons ensuite modifié les applications comme décrit précédemment, puis relancé les outils de notre première contribution sur les applications modifiées pour comparer leur taux de terminaison au taux sur les applications initiales. En fonction des outils, le taux de terminaison est soit inchangé, soit légèrement plus faible pour les applications modifiées.

Pour vérifier que notre approche fonctionne, nous avons créé une petite application de test utilisant du chargement dynamique et des appels réflexifs. Code 16 montre la classe principale de l'application. On peut voir par exemple que l'application utilise des chaînes de caractères chiffrées pour stocker le nom des méthodes à appeler.

APK SHA 256	Nombre d'arcs du graphe d'appel			
	Avant	Après	Différences	Réflexion Ajoutées
0019d7fb6a...	641	60 170	59 529	1
274b677449...	537 613	540 674	3061	26
34599c2499...	336 740	339 616	2876	29
35065c6834...	343 245	346 694	3449	26
e7b2fb02ff...	464 642	465 389	747	91
efececc03c...	243 647	243 925	278	23
f34ce1e7a8...	704 095	706 576	2481	28

Table 19: Arcs ajoutés aux graphes d'appel de fonctions des applications instrumentées, calculé par Androguard

```java
package com.example.theseus;

public class Main {
    private static final String DEX = "ZGV4CjA [...] EAAABEAwAA";
    Activity ac;
    private Key key = new SecretKeySpec("_-_Secret Key_-_".getBytes(),
"AES");
    ClassLoader cl = new
InMemoryDexClassLoader(ByteBuffer.wrap(Base64.decode(DEX, 2)),
Main.class.getClassLoader());

    public void main() throws Exception {
        String[] strArr = {"n6WGYJzjDrUvR9cYljlNlw==", "dapES0wl/
iFIPuMnH3fh7g=="};
        Class<?> loadClass =
this.cl.loadClass(decrypt("W5f3xRf3wCSYcYG7ckYGR5xuuESDZ2NcDUzGxsq3sls="));
        Object obj = "imei";
        for (int i = 0; i < 2; i++) {
            obj = loadClass.getMethod(decrypt(strArr[i]),
String.class, Activity.class).invoke(null, obj, this.ac);
        }
    }
    public String decrypt(String str) throws Exception {
        Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
        cipher.init(2, this.key);
        return new String(cipher.doFinal(Base64.decode(str, 2)));
    }

    ...
}
```

Code 16: Code de la classe principale de l'application calculé par Jadx, avant modification

Après avoir lancé notre analyse statique et instrumenté l'application pour y ajouter les informations dynamique, Jadx montre maintenant le Code 17. On peut voir que les méthodes `Malicious.get_data()` et `Malicious.send_data()` sont appelées. De plus, la classe `Malicious` qui n'était pas présente dans l'application originale est maintenant visible dans l'application modifiée.

Dans le code de `Malicious`, `get_data()` retourne des données d'une source d'information sensible, et `send_data()` exfiltre les données qui lui sont passées. Une fuite d'information devrait donc être détectée par Flowdroid. Lancé sur l'application originale, Flowdroid calcule un graphe d'appel

```
 1  public void main() throws Exception {
 2      String[] strArr = {"n6WGYJzjDrUvR9cYljlNlw==", "dapES0wl/
iFIPuMnH3fh7g=="};
 3      Class<?> loadClass =
this.cl.loadClass(decrypt("W5f3xRf3wCSYcYG7ckYGR5xuuESDZ2NcDUzGxsq3sls="));
 4      Object obj = "imei";
 5      for (int i = 0; i < 2; i++) {
 6          Method method = loadClass.getMethod(decrypt(strArr[i]),
String.class, Activity.class);
 7          Object[] objArr = {obj, this.ac};
 8          obj = T.check_is_Malicious_get_data_fe2fa96eab371e46(method) ?
 9            Malicious.get_data((String) objArr[0], (Activity)
objArr[1]) :
10            T.check_is_Malicious_send_data_ca50fd7916476073(method) ?
11            Malicious.send_data((String) objArr[0], (Activity)
objArr[1]) :
12            method.invoke(null, objArr);
13      }
14  }
```

Code 17: Code de `Main.main()` calculé par Jadx, après modifications

de méthodes contenant 43 arcs, et ne détecte aucune fuite. En revanche, lancé sur l'application modifiée, Flowdroid calcule cette fois un graphe de 76 arsc, et détecte bien la fuite de données.

Bien que nous n'ayons pas pu tester notre approche correctement à grande échelle dû aux limites de notre environnement d'analyse statique, nous avons bien montré qu'il est possible de transmettre des informations dynamiques à des outils d'analyse statique pour améliorer leurs résultats.

Bibliography

[1] A. Abraham, R. Andriatsimandefitra, A. Brunelat, J.-F. Lalande, and V. Viet Triem Tong. 2015. GroddDroid: a gorilla for triggering malicious behaviors. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, October 2015. 119–127. https://doi.org/10.1109/MALWARE.2015.7413692

[2] Boladji Vinny Adjibi, Fatou Ndiaye Mbodji, Tegawendé F. Bissyandé, Kevin Allix, and Jacques Klein. 2022. The Devil is in the Details: Unwrapping the Cryptojacking Malware Ecosystem on Android. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, October 2022. 153–163. https://doi.org/10.1109/SCAM55253.2022.00023

[3] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *13th Working Conference on Mining Software Repositories (MSR)*, May 2016. 468–471.

[4] Radoniaina Andriatsimandefitra and Valérie Viet Triem Tong. 2015. Detection and Identification of Android Malware Based on Information Flow Monitoring. In *2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing*, November 2015. 200–203. https://doi.org/10.1109/CSCloud.2015.27

[5] Radoniaina Andriatsimandefitra, Stéphane Geller, and Valérie Viet Triem Tong. 2012. Designing information flow policies for Android's operating system. In *IEEE International Conference on Communications*, June 2012. IEEE Computer Society, Ottawa, Canada, 976–981. https://doi.org/10.1109/ICC.2012.6364161

[6] Daniel Arp, Michael Spreitzenbarth, Hugo Gascon, Konrad Rieck, Germany Siemens, and Cert Munich. 2014. Drebin: Effective and Explainable Detection of Android Malware in Your Pocket. In *Network and Distributed System Security Symposium*, February 2014. The Internet Society, San Diego, California, USA.

[7] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2013. Instrumenting Android and Java Applications as Easy as abc. In *Fourth International Conference on Runtime Verification*, September 2013. Springer Berlin Heidelberg, Rennes, France, 364–381. https://doi.org/10.1007/978-3-642-40787-1_26

[8] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android

Apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 05, 2014. ACM Press, Edinburgh, UK, 259–269. https://doi.org/10.1145/2666356.2594299

[9] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and Depth-First Exploration for Systematic Testing of Android Apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, Part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, 2013. ACM, 641–660. https://doi.org/10.1145/2509136.2509549

[10] Mario Luca Bernardi, Marta Cimitile, Damiano Distante, Fabio Martinelli, and Francesco Mercaldo. 2019. Dynamic malware detection and phylogeny analysis using process mining. *International Journal of Information Security* 18, 3 (June 2019), 257–284. https://doi.org/10.1007/s10207-018-0415-3

[11] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. 2013. Thresher: Precise Refutations for Heap Reachability. *ACM SIGPLAN Notices* 48, 6 (June 2013), 275–286. https://doi.org/10.1145/2499370.2462186

[12] Amiangshu Bosu, Fang Liu, Danfeng (Daphne) Yao, and Gang Wang. 2017. Collusive Data Leak and More: Large-scale Threat Analysis of Inter-app Communications. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, April 02, 2017. ACM, Abu Dhabi United Arab Emirates, 71–85. https://doi.org/10.1145/3052973.3053004

[13] Kwanghoon Choi and Byeong-Mo Chang. 2014. A Type and Effect System for Activation Flow of Components in Android Programs. *Information Processing Letters* 114, 11 (2014), 620–627. https://doi.org/10.1016/j.ipl.2014.05.011

[14] Yuning Cui, Yi Sun, and Zhaowen Lin. 2023. DroidHook: a novel API-hook based Android malware dynamic analysis sandbox. *Automated Software Engineering* 30, 1 (February 2023), 10. https://doi.org/10.1007/s10515-023-00378-w

[15] Anthony Desnos and Geoffroy Gueguen. 2011. Android: From Reversing to Decompilation. *Black Hat Abu Dhabi* (2011). Retrieved from https://media.blackhat.com/bh-ad-11/Desnos/bh-ad-11-DesnosGueguen-Andriod-Reversing_to_Decompilation_WP.pdf

[16] Yue Duan, Mu Zhang, Abhishek Vasisht Bhaskar, Heng Yin, Xiaorui Pan, Tongxin Li, Xueqiang Wang, and Xiaofeng Wang. 2018. Things You May Not Know About Android (Un)Packers: A Systematic Study based on Whole-System Emulation. In *24th Annual Network and Distributed System Security Symposium*, 2018.

[17] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *9th USENIX Symposium on Operating Systems Design and Implementation*, October 2010. USENIX Association, Vancouver, BC, Canada, 393–407.

[18] Farnood Faghihi, Mohammad Zulkernine, and Steven Ding. 2022. CamoDroid: An Android application analysis environment resilient against sandbox evasion. *Journal of Systems Architecture* 125, (April 2022), 102452. https://doi.org/10.1016/j.sysarc.2022.102452

[19] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. 2012. Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, October 16, 2012. ACM, Raleigh North Carolina USA, 50–61. https://doi.org/10.1145/2382196.2382205

[20] Eva Galperin. 2020. The State of the Stalkerware. January 2020. USENIX Association, San Francisco, CA.

[21] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. 2013. Structural Detection of Android Malware Using Embedded Call Graphs. In *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security*, November 04, 2013. ACM, Berlin Germany, 45–54. https://doi.org/10.1145/2517312.2517315

[22] Dimitris Geneiatakis, Igor Nai Fovino, Ioannis Kounelis, and Pasquale Stirparo. 2015. A Permission Verification Approach for Android Mobile Applications. *Computers & Security* 49, (March 2015), 192–205. https://doi.org/10.1016/j.cose.2014.10.005

[23] Li Gong. 1998. Secure Java class loading. *IEEE Internet Computing* 2, 6 (November 1998), 56–61. https://doi.org/10.1109/4236.735987

[24] Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. 2015. Information Flow Analysis of Android Applications in DroidSafe. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015. The Internet Society.

[25] Yi He, Yacong Gu, Purui Su, Kun Sun, Yajin Zhou, Zhi Wang, and Qi Li. 2023. A Systematic Study of Android Non-SDK (Hidden) Service API Security. *IEEE Transactions on Dependable and Secure Computing* 20, 2 (March 2023), 1609–1623. https://doi.org/10.1109/TDSC.2022.3160872

[26] Johannes Hoffmann, Martin Ussath, Thorsten Holz, and Michael Spreitzenbarth. 2013. Slicing Droids: Program Slicing for Smali Code. In *Proceedings of the 28th Annual ACM*

Symposium on Applied Computing (SAC '13), March 18, 2013. Association for Computing Machinery, New York, NY, USA, 1844–1851. https://doi.org/10.1145/2480362.2480706

[27] Jinseong Jeon, Kristopher K. Micinski, Jeffrey A. Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S. Foster, and Todd Millstein. 2012. Dr. Android and Mr. Hide: Fine-Grained Permissions in Android Applications. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, October 19, 2012. ACM, Raleigh North Carolina USA, 3–14. https://doi.org/10.1145/2381934.2381938

[28] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. 2014. Android Taint Flow Analysis for App Sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, June 12, 2014. ACM, Edinburgh United Kingdom, 1–6. https://doi.org/10.1145/2614628.2614633

[29] Pavel Kriz and Filip Maly. 2015. Provisioning of application modules to Android devices. In *2015 25th International Conference Radioelektronika (RADIOELEKTRONIKA)*, April 2015. 423–426. https://doi.org/10.1109/RADIOELEK.2015.7129009

[30] Li Li, Alexandre Bartel, Tegawende F. Bissyande, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, May 2015. IEEE, Florence, Italy, 280–291. https://doi.org/10.1109/ICSE.2015.48

[31] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2015. ApkCombiner: Combining Multiple Android Apps to Support Inter-App Analysis. In *ICT Systems Security and Privacy Protection*, 2015. Springer International Publishing, Cham, 513–527. https://doi.org/10.1007/978-3-319-18467-8_34

[32] Li Li, Tegawendé F. Bissyandé, Damien Octeau, and Jacques Klein. 2016. DroidRA: taming reflection to support whole-program analysis of Android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*, July 2016. Association for Computing Machinery, New York, NY, USA, 318–329. https://doi.org/10.1145/2931037.2931044

[33] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. 2017. Static Analysis of Android Apps: A Systematic Literature Review. *Information and Software Technology* 88, (2017), 67–95. https://doi.org/10.1016/j.infsof.2017.04.001

[34] Li Li, Tegawendé F. Bissyandé, Yves Le Traon, and Jacques Klein. 2016. Accessing Inaccessible Android APIs: An Empirical Study. In *2016 IEEE International Conference*

on Software Maintenance and Evolution (ICSME), October 2016. 411–422. https://doi.org/10.1109/ICSME.2016.35

[35] Sheng Liang and Gilad Bracha. 1998. Dynamic class loading in the Java virtual machine. *SIGPLAN Not.* 33, 10 (October 1998), 36–44. https://doi.org/10.1145/286942.286945

[36] Shuying Liang, Andrew W. Keep, Matthew Might, Steven Lyde, Thomas Gilray, Petey Aldous, and David Van Horn. 2013. Sound and Precise Malware Analysis for Android via Pushdown Reachability and Entry-Point Saturation. In *Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones & Mobile Devices (SPSM '13)*, November 08, 2013. Association for Computing Machinery, New York, NY, USA, 21–32. https://doi.org/10.1145/2516760.2516769

[37] Yibin Liao, Jiakuan Li, Bo Li, Guodong Zhu, Yue Yin, and Ruoyan Cai. 2016. Automated Detection and Classification for Packed Android Applications. In *International Conference on Mobile Services*, June 2016. IEEE, San Francisco, USA, 200–203.

[38] Max Lillack, Christian Kästner, and Eric Bodden. 2014. Tracking Load-Time Configuration Options. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*, September 15, 2014. Association for Computing Machinery, New York, NY, USA, 445–456. https://doi.org/10.1145/2642937.2643001

[39] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *Proceedings of the 36th International Conference on Software Engineering*, May 31, 2014. ACM, Hyderabad India, 1013–1024. https://doi.org/10.1145/2568225.2568229

[40] Linghui Luo, Felix Pauck, Goran Piskachev, Manuel Benz, Ivan Pashchenko, Martin Mory, Eric Bodden, Ben Hermann, and Fabio Massacci. 2022. TaintBench: Automatic Real-World Malware Benchmarking of Android Taint Analyses. *Empirical Software Engineering* 27, 1 (January 2022), 16. https://doi.org/10.1007/s10664-021-10013-5

[41] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*, July 2016. Association for Computing Machinery, New York, NY, USA, 94–105. https://doi.org/10.1145/2931037.2931054

[42] Noah Mauthe, Ulf Kargén, and Nahid Shahmehri. 2021. A Large-Scale Empirical Study of Android App Decompilation. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, March 2021. 400–410. https://doi.org/10.1109/SANER50967.2021.00044

[43] René Mayrhofer, Jeffrey Vander Stoep, Chad Brubaker, and Nick Kralevich. 2021. The Android Platform Security Model. *ACM Trans. Priv. Secur.* 24, 3 (April 2021), 19:1–19:35. https://doi.org/10.1145/3448609

[44] Jean-Marie Mineau and Jean-Francois Lalande. 2024. Evaluating the Reusability of Android Static Analysis Tools. In *Reuse and Software Quality*, 2024. Springer Nature Switzerland, Cham, 153–170.

[45] Jean-Marie Mineau and Jean-François Lalande. 2025. Class Loaders in the Middle: Confusing Android Static Analyzers. *Digital Threats* 6, 3 (September 2025). https://doi.org/10.1145/3754457

[46] Zhenyu Ning and Fengwei Zhang. 2018. DexLego: Reassembleable bytecode extraction for aiding static analysis. *Proceedings - 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018* (2018), 690–701. https://doi.org/10.1109/DSN.2018.00075

[47] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. 2015. Composite Constant Propagation: Application to Android Inter-Component Communication Analysis. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, May 2015. IEEE, Florence, Italy, 77–88. https://doi.org/10.1109/ICSE.2015.30

[48] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. 2013. Effective Inter-Component communication mapping in android: An essential step towards holistic security analysis. In *22nd USENIX Security Symposium (USENIX Security 13)*, 2013. 543–558.

[49] Felix Pauck, Eric Bodden, and Heike Wehrheim. 2018. Do Android Taint Analysis Tools Keep Their Promises?. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, October 26, 2018. ACM, Lake Buena Vista FL USA, 331–341. https://doi.org/10.1145/3236024.3236029

[50] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. 2018. TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time. (2018).

[51] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2014. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014. The Internet Society.

[52] Zhengyang Qu, Shahid Alam, Yan Chen, Xiaoyong Zhou, Wangjun Hong, and Ryan Riley. 2017. DyDroid: Measuring Dynamic Code Loading and Its Security Implications in Android Applications. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2017. 415–426. https://doi.org/10.1109/DSN.2017.14

[53] Bradley Reaves, Jasmine Bowers, Sigmund Albert Gorski III, Olabode Anise, Rahul Bobhate, Raymond Cho, Hiranava Das, Sharique Hussain, Hamza Karachiwala, Nolen Scaife, Byron Wright, Kevin Butler, William Enck, and Patrick Traynor. 2016. *droid: Assessment and Evaluation of Android Application Analysis Tools. *ACM Computing Surveys* 49, 3 (October 2016), 55:1–55:30. https://doi.org/10.1145/2996358

[54] Atanas Rountev and Dacong Yan. 2014. Static Reference Analysis for GUI Objects in Android Software. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, February 15, 2014. ACM, Orlando FL USA, 143–153. https://doi.org/10.1145/2544137.2544159

[55] Antonio Ruggia, Dario Nisi, Savino Dambra, Alessio Merlo, Davide Balzarotti, and Simone Aonzo. 2024. Unmasking the Veiled: A Comprehensive Analysis of Android Evasive Malware. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security (ASIA CCS '24)*, July 2024. Association for Computing Machinery, New York, NY, USA, 383–398. https://doi.org/10.1145/3634737.3637658

[56] Jordan Samhi, Jun Gao, Nadia Daoudi, Pierre Graux, Henri Hoyez, Xiaoyu Sun, Kevin Allix, Tegawendé F. Bissyandé, and Jacques Klein. 2022. JuCify: a step towards Android code unification for enhanced static analysis. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*, July 2022. Association for Computing Machinery, New York, NY, USA, 1232–1244. https://doi.org/10.1145/3510003.3512766

[57] Zhiyong Shan, Iulian Neamtiu, and Raina Samuel. 2018. Self-Hiding Behavior in Android Apps. In *40th International Conference on Software Engineering*, 2018. ACM Press, New York, New York, USA, 728–739. https://doi.org/10.1145/3180155.3180214

[58] Feng Shen, Namita Vishnubhotla, Chirag Todarka, Mohit Arora, Babu Dhandapani, Eric John Lehner, Steven Y. Ko, and Lukasz Ziarek. 2014. Information Flows as a Permission Mechanism. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, September 15, 2014. ACM, Vasteras Sweden, 515–526. https://doi.org/10.1145/2642937.2643018

[59] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis.

In *2016 IEEE Symposium on Security and Privacy (SP)*, 2016. 138–157. https://doi.org/10.1109/SP.2016.17

[60] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*, August 2017. Association for Computing Machinery, New York, NY, USA, 245–256. https://doi.org/10.1145/3106237.3106298

[61] Thomas Sutter, Timo Kehrer, Marc Rennhard, Bernhard Tellenbach, and Jacques Klein. 2024. Dynamic Security Analysis on Android: A Systematic Literature Review. *IEEE Access* 12, (2024), 57261–57287. https://doi.org/10.1109/ACCESS.2024.3390612

[62] Kimberly Tam, Salahuddin Khan, Aristide Fattori, and Lorenzo Cavallaro. 2015. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In *22nd Annual Network and Distributed System Security Symposium*, February 2015. The Internet Society, San Diego, California, USA.

[63] Dennis Titze and Julian Schutte. 2015. Apparecium: Revealing Data Flows in Android Applications. In *2015 IEEE 29th International Conference on Advanced Information Networking and Applications*, March 2015. IEEE, Gwangiu, South Korea, 579–586. https://doi.org/10.1109/AINA.2015.239

[64] Akihiko Tozawa and Masami Hagiya. 2002. Formalization and Analysis of Class Loading in Java. *Higher-Order and Symbolic Computation* 15, 1 (March 2002), 7–55. https://doi.org/10.1023/A:1019912130555

[65] Timothy Vidas, Jiaqi Tan, Jay Nahata, Chaur Lih Tan, Nicolas Christin, and Patrick Tague. 2014. A5: Automated Analysis of Adversarial Android Applications. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, November 07, 2014. ACM, Scottsdale Arizona USA, 39–50. https://doi.org/10.1145/2666620.2666630

[66] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2014. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *ACM SIGSAC Conference on Computer and Communications Security*, November 2014. ACM, Scottsdale Arizona USA, 1329–1341. https://doi.org/10.1145/2660267.2660357

[67] Erik Ramsgaard Wognsen, Henrik Søndberg Karlsen, Mads Chr. Olesen, and René Rydhof Hansen. 2014. Formalisation and Analysis of Dalvik Bytecode. *Science of Computer Programming* 92, (October 2014), 25–55. https://doi.org/10.1016/j.scico.2013.11.037

[68] Michelle Y Wong and David Lie. 2018. Tackling runtime-based obfuscation in Android with TIRO. In *USENIX Security Symposium*, August 2018. USENIX, Baltimore, USA, 1247–1262.

[69] Mingyuan Xia, Lu Gong, Yuanhao Lyu, Zhengwei Qi, and Xue Liu. 2015. Effective Real-Time Android Application Auditing. In *2015 IEEE Symposium on Security and Privacy*, May 2015. IEEE, San Jose, CA, 899–914. https://doi.org/10.1109/SP.2015.60

[70] Lei Xue, Xiapu Luo, Le Yu, Shuai Wang, and Dinghao Wu. 2017. Adaptive unpacking of Android apps. In *International Conference on Software Engineering*, May 2017. IEEE, Buenos Aires, Argentina, 358–369.

[71] Lok Kwong Yan and Heng Yin. 2012. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *21st USENIX Security Symposium (USENIX Security 12)*, August 2012. USENIX Association, Bellevue, WA, 569–584. Retrieved from https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/yan

[72] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. 2015. Static Control-Flow Analysis of User-Driven Callbacks in Android Applications. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, May 2015. IEEE, Florence, Italy, 89–99. https://doi.org/10.1109/ICSE.2015.31

[73] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Junliang Shu, Bodong Li, Wenjun Hu, and Dawu Gu. 2015. AppSpear: Bytecode Decrypting and DEX Reassembling for Packed Android Malware. In *Research in Attacks, Intrusions, and Defenses (Lecture Notes in Computer Science)*, 2015. Springer International Publishing, Cham, 359–381. https://doi.org/10.1007/978-3-319-26362-5_17

[74] Yueqian Zhang, Xiapu Luo, and Haoyang Yin. 2015. Dexhunter: toward extracting hidden code from packed android applications. In *European Symposium on Research in Computer Security*, November 2015. Springer, Vienna, Austria, 293–311.

[75] Yury Zhauniarovich, Maqsood Ahmad, Olga Gadyatskaya, Bruno Crispo, and Fabio Massacci. 2015. 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*, March 02, 2015. ACM, San Antonio Texas USA, 37–48. https://doi.org/10.1145/2699026.2699105

[76] Wenwen Zhou, Yang Yongzhi, and Jiejuan Wang. 2022. Dynamic Class Generating and Loading Technology in Android Web Application. In *2022 International Symposium on Networks, Computers and Communications (ISNCC)*, July 2022. 1–6. https://doi.org/10.1109/ISNCC55209.2022.9851782

COLLEGE MATHS, TELECOMS
DOCTORAL INFORMATIQUE, SIGNAL
BRETAGNE SYSTEMES, ELECTRONIQUE

CentraleSupélec

Titre : Les difficultés de la rétro-ingénierie Android: de l'analyse large échelle au dé-brouillage dynamique

Mots clés : Android, Analyse de Maliciels, Ingénierie Inverse, Chargement de Classe, Brouillage de Code

Résumé : La place croissante des téléphones mobiles dans notre vie quotidienne en font une cible de choix pour les acteurs malveillants. Cette menace rend l'analyse d'application cruciale pour déterminer s'il s'agit ou non d'un maliciel et de son impact sur l'utilisateur s'il s'agit bien d'une application malveillante.

Cette thèse explore les difficultés liées à l'ingénierie inverse d'applications Android. Dans un premier temps, elle reprend un effort de la communauté qui a identifié les contributions entre 2011 et 2017 portant sur l'analyse statique d'applications mobiles et propose une méthode pour évaluer la réutilisabilité des outils associés. Une étude poussée des échecs lors de l'exécution de ces outils montre que 54.55% d'entre eux ne sont plus utilisables. Elle modélise ensuite le processus de chargement des classes utilisées par une application et présente une méthode de brouillage basée sur les différences entre ce modèle et celui utilisé par des outils d'analyses tels que Androguard ou Flowdroid. Enfin, elle propose une approche consistant à encoder les résultats d'une analyse dynamique dans une nouvelle application valide pour permettre aux outils existants d'analyser des applications faisant usage de chargement de code dynamique.

Title : The Woes of Android Reverse Engineering: from Large Scale Analysis to Dynamic Deobfuscation

Keywords: Android, Malware Analysis, Reverse Engineering, Class Loading, Code Obfuscation

Abstract: The growing importance of smartphones in our daily lives makes them a prime target for malicious actors. This makes our ability to analyse an application critical, as it allows us to determine if an application is malware and its impact on the user.

This thesis explores the difficulties of reverse engineering an Android application. First, we pursue a community effort that identified contributions between 2011 and 2017 about static analysis of Android applications, and we propose a method to evaluate the reusability of the associated tools. An extensive analysis of the execution failures of those tools shows that 54.55% of them are no longer usable. Then, we model the mechanism that loads the classes used by an application and present an obfuscation method based on the discrepancies between this model and the one used by analysis tools like Androguard and Flowdroid. Finally, we propose an approach that consists of encoding the result of a dynamic analysis within a new valid application, allowing existing tools to analyse applications that rely on dynamic code loading.