University of Brasília

Exact Sciences Institute
Computer Science Department

The use of Argument Comparison to Improve the Performance of Mining Sandboxes for Android Malware Detection

João Victor de Souza Calassio

Monograph submitted in partial fullfilment of
the requirements to Bachelor Degree in Computer Science

Advisor
Prof. Dr. Rodrigo Bonifácio de Almeida

Brasília
2023

University of Brasília

Exact Sciences Institute
Computer Science Department

The use of Argument Comparison to Improve the Performance of Mining Sandboxes for Android Malware Detection

João Victor de Souza Calassio

Monograph submitted in partial fullfilment of
the requirements to Bachelor Degree in Computer Science

Prof. Dr. Rodrigo Bonifácio de Almeida (Advisor)
CIC/UnB

Francisco Handrick da Costa
University of Brasília

Ismael Medeiros
University of Brasília

Prof. Dr. Marcelo Grandi Mandelli
Coordinator of Bachelor Degree in Computer Science

Brasília, July 24, 2023

Dedicated to

I would like to dedicate this work primarily to my mother, Vilma. She have always known the impact that education has on the life of a person and the ones around them. Ever since I was a child, I have always been incentivized to dedicate myself, and told that by doing that I would have a bright future. In fact, she have always done everything that she could to provide me with the best condition to achieve that future, and I certainly wouldn't be here if it wasn't for all of her efforts. I will forever be grateful.

I would also like to thank for my dearest friends, Diego, Liverson and Lucius that have always been with me since school, and also thank the wonderful friends that I made on the way: Adelson, Breno, Carlos, Gabriel, João Pedro, José, Matheus, Vinicius, Yuri, William, and many others.

With deepest gratitude,

João Victor.

Acknowledgements

I would like to express my deepest gratitude to my advisor professor Rodrigo Bonifácio, for his guidance and encouragement during the development of this research, and Ismael Medeiros for joining my defense committee and contributing with their knowledge.

I'm also extremely grateful to Francisco Handrick da Costa, for his support and expertise on the subject, and the opportunity to contribute on his doctoral thesis.

Lastly, I'd also like to thank the University of Brasilia for providing me the opportunity to conduct this research and complete my Bachelor's degree of Computer Science.

Abstract

The Android platform, with its extensive user base and popularity has become a prime target for malware attacks. For that reason, researchers have been interested on malware detection methods, including the *mining sandboxes* approach. This approach focuses on *repackaged* apps, a type of attack that consists on modifying an existing app and introducing malicious behavior on it, and is highly prevalent on the Android platform. The mining sandboxes technique takes advantage of test case generation tools to monitor an app's runtime behavior and detect potential malicious intentions through behavioral differences between different versions of apps. While the studies have shown promising conclusions, with over 70% detection accuracy, there's still a lot of room for improvement.

This study investigates how the performance of the mining sandboxes can be improved by combining some previously proposed techniques (such as static analysis) with an approach that extends the behavioral differences detection by taking into consideration the arguments passed to sensitive methods, and if the type of malware has any influence on the detection effectiveness. This is done by evolving DroidXP, an existing research framework for mining sandboxes, and evaluating its performance on a comprehensive dataset of 1,707 pairs of apps. The results show that there's an improvement of 14% on the malware detection accuracy, and that there's a high influence of the type of malware on the detection outcome.

Keywords: mining sandboxes, android malware detection, dynamic analysis

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Mobile devices have become a huge part of our lives, offering an extensive set of functionalities such as internet access, banking, high quality camera and microphones, global location services, and much more. The Android platform is the most used mobile platform in the world, with 71.63% market share and 3.6 billion users worldwide [3]. Android applications, commonly known as *apps*, are distributed in various marketplaces, with Google Play Store being the largest and most used having over 3.5 million apps available [4]. The huge popularity of mobile devices combined with the access to many sensitive capabilities have turned mobile devices on attractive targets for malicious actors. In fact, Truong et al. have reported that over 0.25% of Android devices were infected with malware [5].

Most Android malware employ a type of attack called *repackaging*, which consists of altering an existing application to introduce malicious behavior. These repackaged apps are distributed in various marketplaces, and trick users into believing that they're downloading the original version of the app. Once installed on a device, these harmful applications are capable of capturing user input, recording data from the microphone or camera and sending it to third-party servers.

To address this issue, previous works studied a method called *mining sandboxes*, first proposed by Jamrozik et al. [6] that consists on capturing an app behavior using automated test generation tools. Bao et al. demonstrated that mining sandboxes are an effective mechanism to detect malicious activity on Android apps, and also compared which test case generation tools produced the best results for this purpose [7]. Building up on Bao et al's study, Costa et al. [8] developed a tool called DroidXP that combines static analysis with the dynamic analysis proposed by the mining sandboxes approach to achieve a higher effectiveness on malware detection. Le et al. [9] have also extended Jamrozik's work by creating a more robust sandbox that not only considers new behaviors introduced by repackaged versions of the apps, but also identifies differences on existing behaviors, such as different arguments passed to sensitive methods.

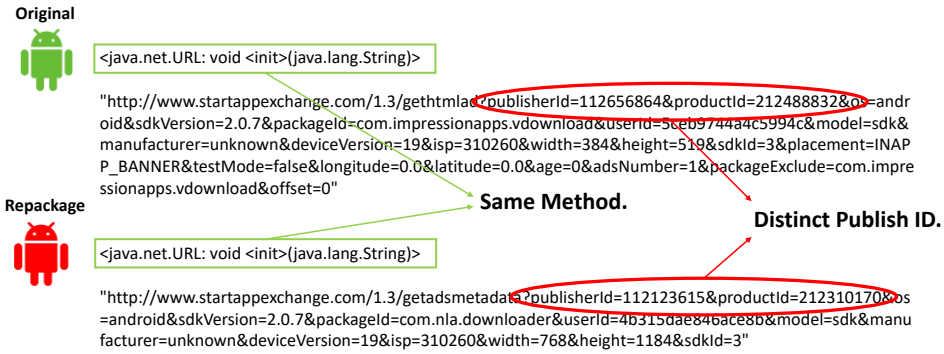

Figure 1.1: Example of a successful detection comparing the arguments passed to sensitive methods. In this example, the internet connection API is used to access a advertisement URL. The advertisement URL is changed to redirect ad revenue to the publisher of the repackaged version of the app.

In this study, we aim to develop the existing technique by conducting a more comprehensive study, where we will evaluate the performance of combining Costa's approach of considering a static analysis along with the dynamic analysis [8], with Le's proposal of comparing arguments passed into sensitive APIs [9] to detect Android repackaged malware. In addition, previous studies have utilized small datasets, with 102 pairs of apps for [7] and [8], and 25 pairs for [9], which may compromise the external validity of the findings. To address this issue, we will use a significantly larger dataset consisting of 1,707 pairs. In general, to achieve the goals of this study, we adapted the DroidXP framework [2], so that it could be able to instrument Android apps and capture the arguments passed to a given set of methods.

The primary contribution of this work lies on the evolution of the DroidXP framework and the evaluation of this new approach regarding the accuracy of its malware detection capabilities. Our results indicate that there is an improvement of 14% on the overall accuracy if compared to the previous version, and the detection capabilities of this new approach are highly dependant on the malware family. While some families show a substantial improvement in detection rates, others express no relevant change.

Chapter 2

Background

2.1 Android architecture

Android is an open source operating system, based on the Linux kernel and created to fit multiple types of devices [10]. Android applications (or *apps*) are written using programming languages that compile for the Java Virtual Machine (JVM) bytecode, such as Java or Kotlin. The JVM bytecode is compiled to a Dalvik Executable (DEX) bytecode format, and is then packaged along with other application resources to form an Android Application Pack (APK) file, which can be distributed for end-users. The app building process can be seen at Figure 2.1.

The Android platform identifies and isolate each individual app and its resources, by assigning them a unique identifier (UID) and running each Android app on its own process [11] of the Android Runtime virtual machine, which interprets the DEX bytecode. The assigned UID is used to establish a kernel-level application sandbox, that isolates every app from each other, protecting both the apps and the system from malicious activity.

In order to access information beyond the application sandbox, apps must acquire permissions and then access the resources through specific system Application Programming Interfaces (APIs). Android has two main types of permissions [12]: install-time permissions, and runtime permissions. Install-time permissions are less sensitive permissions and must be requested upon installation of the APK on the system, such as internet access. Runtime permissions are highly sensitive permissions and are generally related to users' private data, such as camera or microphone access. Runtime permissions must be acquired right before the actual use of the resource, and are only available on Android 6.0 or higher.

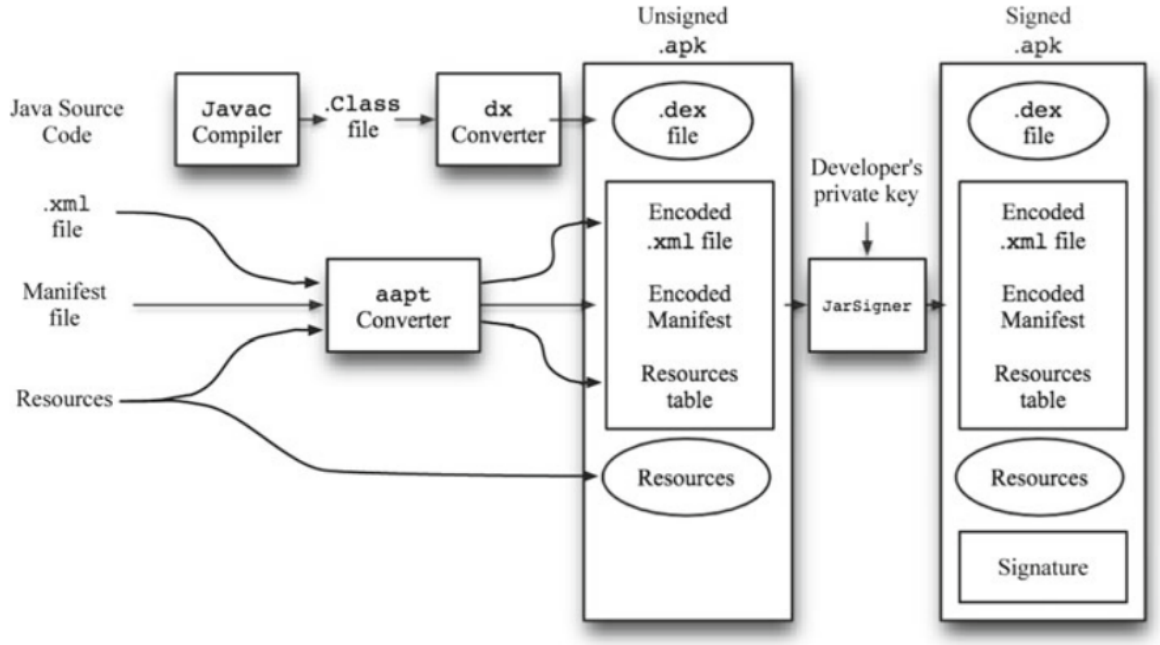

Figure 2.1: Android build process, from source code to APK file. Source: [1].

2.2 Android malwares

A *malware* is a type of application that's capable of executing code that may carry out harmful activities on a system, its modules, or other applications. It's defined as a "set of instructions that run on your computer and make your system do something that an attacker wants it to do" [13]. Mobile devices have evolved to perform activities that were only possible on desktop computers, such as internet access or banking, along with having its own set of features such as calling, SMS, access to precise location, or high quality cameras and microphones and for that reason they are a huge target for malware and other types of malicious attacks.

Zhou et al. [14] characterizes Android malware based on their installation, activation and the carried malicious payloads. The techniques used by the malware to install onto the user devices were analyzed, and separated on three main groups: *repackaging*, *update attack* and *drive-by download*, which aren't necessarily mutually exclusive. Regarding the activation mechanisms, the existing Android events were examined to determine how malware used them to flexibly trigger their activation (e.g., only starting its background services when the system is started). Finally, the payloads were also analyzed and classified in four major categories: *privilege escalation*, *remote control*, *financial charges*, and *personal information stealing*.

Although the isolation provided by the application sandbox and restrictions enforced by the permission mechanism add a layer of protection for Android devices, they are not

flawless. Felt et al. revealed that around 33% of Android apps are overprivileged and the developers fail to achieve the least privilege principle [15]. Moreover, a portion of the users do not pay enough attention to the required permissions of the apps they install [16].

2.3 Repackaging malwares

Repackaging is one type of attack on the Android ecosystem, and [14] have shown that the majority (around 86% on the analyzed dataset) of Android malware make use of this type of attack. It consists on gathering an existing application, reverse engineering its code, inserting a malicious activity and then publishing the modified (i.e. repackaged) application on app marketplaces. Once a repackaged app is installed on an Android device and is granted the requested permissions, malicious actors can then collect private user data, perform malicious activities and exploit any other actions allowed by the granted permissions.

Since the application code is compiled to a DEX bytecode, some characteristics about the original code such as class names, method names, and types are retained, which makes them more susceptible for reverse engineering if compared to native code [17].

2.4 The Mining Android Sandbox (MAS) technique

The mining sandboxes approach for the Android platform was first proposed by Jamroski et al. and consists of exploring the behaviour of software by extracting the accessed resources through an automated test generation, and then constructing a sandbox based on those resources [6]. This "mined" sandbox would then prevent and detect unexpected behaviour changes, such as latent malware, infections, malicious updates, etc.

Jamroski et al. introduced a tool called *Boxmate*, that performs such mining sandbox technique, and creates a resulting sandbox much more fine-grained than the original sandbox present on the Android operating system. Boxmate uses a tool called *Droidmate* as test generation tool. However, this study only evaluated the effectiveness of this technique on benign apps, and didn't investigate if it was effective in detecting malware.

This study was then complemented by Bao et al., that evaluated the effectiveness of the mining sandboxes technique on detecting malicious behaviours on Android apps, and investigated the effectiveness of test generation tools for mining sandboxes [7]. It makes use of a set of pairs of apps where each pair consist of a benign app and a malicious repackaged version of the same app. Bao et al.'s study shows that up to 75.5% of the malicious versions of the apps can be identified using the mining sandboxes approach. It

also shows that DroidBot [18] is the most effective test case generation tool among the tested options, which included Monkey, Droidmate, Droidbot, GUIRipper, and PUMA.

2.5 DroidFax

DroidFax is an open-source toolkit that instruments Android apps and performs a series of analyses (both static analysis and dynamic analysis through test case generation tools) on the application execution structure and sensitive data accesses at runtime. It was developed by Cai H. to perform a study on Android apps runtime characteristics and their security implications [19].

Costa et al. [8] investigated the impact of DroidFax's static analysis algorithms for malware identification, and how it can complement the malware detection capabilities of the MAS approach presented by Bao et al.'s study [7]. Costa et al.'s study have shown that 43% of the malware could be detected using the static analysis alone.

DroidFax instrumentation runs on top of Soot [20, 21], a framework to analyze, instrument and optimize Java bytecode. The instrumentation is made by transforming the bytecode in an intermediate representation (Jimple, in the case of DroidFax), analyzing and manipulating (by adding logs, for example) the source code, and finally producing the modified bytecode.

2.6 DroidXP

DroidXP is a benchmarking suite developed by Costa et al. at [2] and its initial goal was to provide a software infrastructure that's capable of comparing the performance of multiple test case generation tools on mining sandboxes for the Android platform. It relies on DroidFax as instrumentation and static analysis tool, and supports many test case generation tools out of the box for the dynamic analysis (and can be easily extended to support other tools). The benchmark is performed on three main phases, as illustrated by Figure 2.2, which are performed after the user specifies the test case generation tool, the number of repetitions for every app version, and the test execution period:

1. Phase one – Instrumentation: in this phase, DroidXP makes use of DroidFax to perform the instrumentation and static analysis on the app pairs provided by the user.

2. Phase two – Execution: then, DroidXP installs the instrumented apps on an Android emulator and runs the test case generation tool for the specified period of time, for every specified repetition. During the execution, all the logs produced

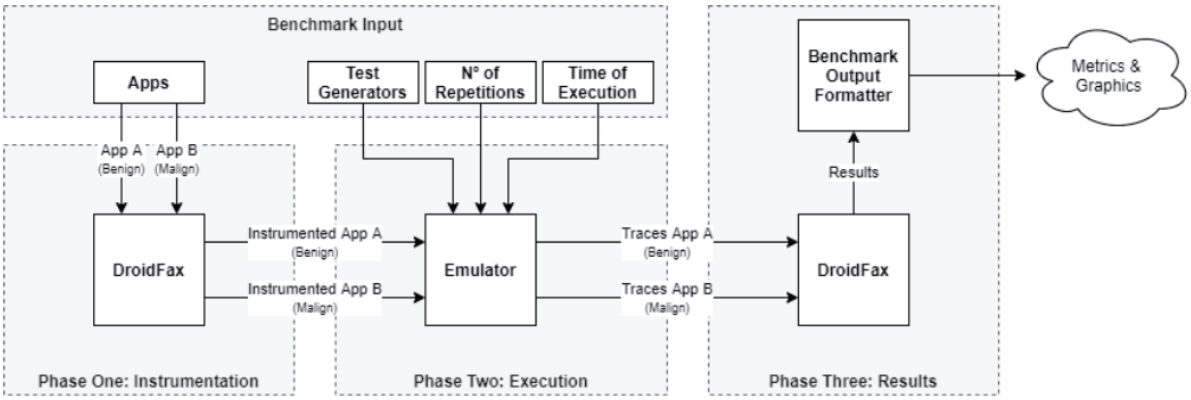

Figure 2.2: DroidXP benchmark architecture. Source: [2].

by the DroidFax dynamic analysis is collected using Android SDK's logcat [22]. The emulator is fully reset after every app execution, to ensure that no executions interfere with each other.

3. Phase three – Report: finally, DroidXP analyzes the results of both static and dynamic analysis and generates the experiment results, which includes the code coverage of the test case generation tool and the sensitive Android APIs accessed by each version of each app.

DroidXP further improved by Costa et al. [8], to investigate the impacts of using both static and dynamic analysis for Android malware identification.

Chapter 3

Implementation details

In this chapter, the implementation details of this new version of DroidXP (and it's underlying engine, DroidFax) will be disclosed, along with the tool used to calculate the differences between executions.

3.1 DroidFax and Soot instrumentation

The current implementation of DroidXP makes use of DroidFax as a tool to instrument and perform static analysis on Android applications. DroidFax main instrumentation process is called `dynCG` (an abbreviation for "dynamic call graph"), which is responsible for building a call graph of a given set of methods when performing the dynamic analysis. It also supports attaching some other instrumentation modes in this process, such as the so called "coverage tracker", "event tracker" and "intent tracker".

In order to capture sensitive method invocations with its arguments, it was necessary to adapt DroidFax[1], by creating a new attachable module to run with the main process. This new module was called "API Tracker", and was built following the patterns of DroidFax's other instrumentations. It consists of three main classes: `Monitor`, `Options` and `sceneInstr`.

The main purpose of the API Tracker instrumentation is to detect every method invocation that is a potential data sink and insert a piece of code that's capable of capturing the arguments passed to these methods at runtime. These sensitive method invocations would then be captured by DroidXP when performing the execution phase, and would be used to detect potential malicious activity (such as a data leakage).

[1] `https://github.com/droidxp/droidfax-fork`

The observed methods were the same set of methods that build the call graph, with a few additions. The method signatures were stored in a text file[2], and were loaded during the instrumentation.

3.1.1 Options class

The `Options` class is an utility class, mainly used to parse parameters passed to the DroidFax executable. The only relevant addition to this file, compared to other `Options` files from the other modules is the `catsink` variable, that's used to store the URI path of the text file containing the methods to be tracked.

3.1.2 Monitor class

The `Monitor` class is a class that's injected into the final Android app. It implements an `apiCall` method, that receives a string and an array of Java objects.

The string represents the signature of the method that's going to be called. The Java objects represent each argument that is passed to this method, ordered. Given that the arguments must be passed as an array of `Object`, every Java primitive type must be converted into it's boxed equivalent before being inserted into the array.

The argument array is converted into a comma separated string, by transforming each object into it's string representation (every Java `Object` implement a `toString()` method) and escaping line breaks and double quotes. Null values are replaced by the "null" string, to avoid null pointer exceptions.

The method and the string representing the arguments are then logged using Android's own logging API, so they can be retrieved by DroidXP.

The `methodSignature` string expects the signature of the method that was called.

The `params` array expects an array of Java objects, where each item represent an argument that was passed to the method at `methodSignature`.

3.1.3 sceneInstr class

The `sceneInstr` class is the main executable class. It's responsible for traversing the classes and methods of the provided APK using Soot to detect the sensitive API calls, and insert the `Monitor.apiCall` call before the actual method invocation. It's also where the CLI arguments are parsed and inserted into the `Options` class. It is built upon the base structure of other DroidFax's `sceneInstr` classes.

In the following sections, the relevant methods of this class will be disclosed.

[2] `https://github.com/droidxp/droidfax-fork/blob/master/catsinks.txt.final`

```
1   private static void apiCall_impl(String methodSignature,
2                                    Object[] params) {
3       try {
4           String args = "";
5           for (Object param : params) {
6               if (param == null) {
7                   args += "null" + ", ";
8               } else {
9                   args += "\"" + param.toString()
10                      .replaceAll("\"", "\\\\\"")
11                      .replaceAll("\n", "\\\\\\\\n") + "\", ";
12              }
13          }
14
15          if (args.length() >= 2) {
16              args = args.substring(0, args.length() - 2);
17          }
18
19          android.util.Log.i("apicall-monitor",
20              "[API-TRACK]: " + methodSignature + " => " + args);
21      } catch (Exception e) {
22          e.printStackTrace();
23      }
24  }
```

Figure 3.1: Source code of the implementation of the *apiCall* method.

The `run()` method

This method is responsible for being the entry point of the instrumentation process. It calls the `init()` and `instrument()` methods described below.

The `init()` method

This method is responsible for loading the method signatures from the `catsink` file, and associating them with their corresponding Android method category. The methods are then stored into an in-memory list, to be used by the instrumentation process.

It's also in this method that the `Monitor` class is imported into the application package, and a reference for the `apiCall` method is created.

The `instrument()` method

This method is responsible for the actual instrumentation process. The first step is collecting all the classes in the application, and filtering them according to a defined pattern. We only want to capture method calls that are present in the original application code, and ignore the methods introduced by our own instrumentation (such as the `apiCall` method).

After the classes are collected, we iterate over them checking their methods. If the body of the method contains an invocation of any method that's in our `catsink` list, an invocation for the `apiCall` tracker method is added using Soot.

This procedure is made by:

1. Collecting the number of arguments passed to the relevant method:

   This is done by using Soot's `getArgCount()` method.

2. Creating an empty array of Java objects:

   Soot's `generateLocal` method creates a local variable into the inspected method's body. This variable is typed as an array of `Object`, and is initialized using a `newArrayExpr`, that creates an empty array with a fixed length. The length of the array is equal to the number of arguments passed to the method that's being tracked.

   All those steps are performed by creating Soots' statements that will be later inserted into the actual method body. An example of statement creation using Soot can be seen at Figure 3.2.

3. Iterating through the arguments passed to the relevant method:

After the array is created, it's necessary to assign the values for their corresponding array positions. Every argument is a Soot `Value` that can be referenced in an array position.

However, not every argument will be assignable to an `Object`. If a variable of a primitive type is passed to a Java `Object` type, a JVM error will be raised and the application will stop running. For this reason, it was necessary to convert primitive types to their corresponding boxed types.

Every Soot `Value` instance have a *type* property. This type is a subclass of a generic `Type` class, such as `PrimType`, `RefLikeType`, etc.. For every argument, we check if their type is an instance of `PrimType`, and if they are, we can find a `RefLikeType` that represents the boxed type using the `PrimType.boxedType()` method.

After discovering the corresponding boxed type, it's necessary to perform the conversion. Firstly, a new local variable typed as the boxed type is generated, and then a verification is made to ensure that this type implements a `valueOf` method receiving the primitive type as an argument. By default, every Java boxed type should implement this method, but the check is made to ensure that the instrumentation won't fail.

After we're sure that the class implements the `valueOf` method, a new invoke expression statement is created, along with the assignment statement for the created variable. The boxed value is now ready to be inserted into our array of objects, and this is also done by creating a new Soot statement.

4. Create the invocation of the `apiCall` method

   It's also necessary to create a statement to invoke our `apiCall` method, that will log the tracked method and its collected arguments. This is done by creating a new `invokeStmt` using the `Monitor.apiCall` method reference (created before the instrumentation began) and passing the relevant method signature as a string, along with the newly created local variable representing the array of objects.

5. Add the new statements to the method body

   After all the new statements are created, the last step is to add them to the method body. It's done by using Soot's `insertBeforeNoRedirect` method, where a list of statements is passed to be inserted before an existing statement (in this case, before the relevant method invocation). A code example can be seen on listing 3.3.

After this process is done and every method from every class is analyzed (and instrumented, if suitable), the instrumentation process is completed.

```
1   // sMethod is any instance of SootMethod
2   Body body = sMethod.retrieveActiveBody();
3   LocalGenerator bodyGenerator = new LocalGenerator(body);
4   List<Stmt> stmtList = new ArrayList<Stmt>();
5
6   // create Object[] variable (uninitialized)
7   Local arrLocal = bodyGenerator
8       .generateLocal(ArrayType.v(
9           RefType.v("java.lang.Object"), 1)
10      );
11
12  int arrayLength = 3;
13  // create new Object[3] initialization expression
14  NewArrayExpr arrExpr = Jimple
15      .v()
16      .newNewArrayExpr(
17          RefType.v("java.lang.Object"),
18          IntConstant.v(arrayLength)
19      );
20
21  // assign the initialization to the variable
22  AssignStmt assignArrToLocal = Jimple
23      .v()
24      .newAssignStmt(arrLocal, arrExpr);
25
26  // add the Object[] arrLocal = new Object[3] to
27  // our to-be-inserted statement list
28  stmtList.add(assignArrToLocal);
```

Figure 3.2: Local variable creation, along with initialization statement and assignment statement using Soot

```
1   PatchingChain<Unit> pchn = body.getUnits();
2
3   Iterator<Unit> itchain = pchn.snapshotIterator();
4   while (itchain.hasNext()) {
5       // get current statement that's being analyzed
6       Stmt s = (Stmt) itchain.next();
7
8       // add other statements that should be inserted
9       // in the current body
10      stmtList.add( ... );
11      stmtList.add( ... );
12
13      InstrumManager.v().insertBeforeNoRedirect(pchn, stmtList, s);
14  }
```

Figure 3.3: Statements insertion before an existing statement using Soot

```
1       adb shell am broadcast -a \
2       android.intent.action.BOOT_COMPLETED
```

Figure 3.4: Android Debug Bridge command command used to broadcast an Android intent through the system's activiy manager.

3.2 DroidXP changes

After the changes in the DroidFax code, it was necessary to perform a few adjustments on DroidXP[3] code. Those changes were focused on enabling the newly created instrumentation, and parsing its results to a Comma Separated Values (CSV) format.

Firstly, it was necessary to add an optional parameter in the DroidXP's CLI that enables our argument catching instrumentation. It was chosen to be `-p`.

Then, if this flag was activated, it would pass the `-monitorApiCalls` argument for the instrumentation at the first phase of the execution (instrumentation).

At the second phase of the execution, a reboot simulation after the APK is installed on the emulator was also added. This was done to try to detect malware that had a flexible activation mechanism based on the device reboot. This reboot is done by issuing a specific Android intent, using the command at 3.4. Also at the second phase, Android ADB's logcat [22] starts listening for any logs tagged with "apicall-monitor", and saves them to a specific file for further parsing and analysis.

At the third phase of the execution, the logs captured in the logcat file for each executed APK are correctly parsed and converted to the CSV format.

[3] `https://github.com/droidxp/benchmark`

```
1   adb wait-for-device shell \
2       `while [[ -z $(getprop sys.boot_completed) ]]; \
3       do sleep 1; done;'
```

Figure 3.5: Android Debug Bridge command command used to wait for the device boot to complete.

It's also important to notice that some changes were made on the method that checks if the emulator has finished booting: in the previous version, sometimes the device wasn't ready yet when the countdown started. It was problematic on some executions, and a new command was added to check for a specific system property. The exact command is present on Figure 3.5.

With all the changes, it was possible to run the tool with a custom parameter, perform the newly implemented instrumentation process and collect the arguments passed for the sensitive methods of each analyzed app. The results were stored in the same way they were stored before, along with a new file called "sensitiveMtdArgs.csv", containing the methods and arguments that were captured during the execution of the application.

3.3 Report generator

After DroidXP's execution is finished, we're left with a lot of CSV files containing the sensitive methods called by both the malicious and benign versions of each app, along with the arguments passed to these methods. In order to compare the execution results, a Python script was created to parse the CSV files for each app version and compare the arguments. We call this script "run assessment"[4].

The assessment is made by firstly grouping all the executions of the same version of the app in a HashSet, a data structure that only allows unique items. This is made to ensure that eventual execution differences will be minimized, and repetitions won't be counted. For each pair of apps, this result in two HashSets, one containing the sensitive methods and the arguments passed for them in the malicious APK execution, and another one with the same structure but for the benign APK execution. Methods that are called more than once, but with different arguments are two separate entries in the HashSet.

Then, the difference between those two sets is calculated, and we are left with a set of every method call that's executed in the malicious version, but is not executed in the benign version. The resulting method calls are filtered to only consider methods that are in a list of "sensitive methods", and the final result is saved into a CSV file.

[4] `https://github.com/droidxp/run-assessment-param`

Chapter 4

Empirical Study

4.1 Study settings

The main goal of this empirical study is to investigate if the performance of the Mining Android Sandboxes technique for malware identification can be improved by capturing and comparing the arguments passed to sensitive API calls. The MAS approach have already shown its potential for malware identification on Bao et al.'s study [7], and Costa et al. have shown that it can be improved by combining it with static analysis tools [8].

This research was performed alongside with an ongoing doctoral thesis that analyzed the combination of both static and dynamic analysis proposed at [8] approach on a larger dataset of 1,707 pairs and the influence of the malware family on the detection.

This dataset was derived from a bigger curated dataset of 15,297 pair of original/repackaged Android apps extracted from the Androzoo repository [23], and were filtered considering a random sampling of 5,700 pairs. From there, 3,381 samples were removed because of incompatibilities with either DroidFax instrumentation or the used Android SDK version. The remaining 2,319 samples had their APKs that were labeled as "benign" submitted to VirusTotal [24], a service that analyzes files and URLs and detects malicious content using over 70 antivirus engines [25], and any pair that VirusTotal accused the benign version to be a malware were removed, resulting in 1,707 pairs. This last step was necessary because the MAS approach assumes that the benign version of the app is actually legitimate.

The ongoing study also only considers the malicious version to actually be malicious when at least two antivirus engines used by VirusTotal label it as malicious. This was the case for 490 pairs.

The 1,707 pairs of apps were submitted to the new version of DroidXP, and the sensitive methods called were captured (with and without the arguments), to determine

if the proposed approach of verifying the arguments passed to sensitive API calls can improve the accuracy of the malware detection.

After the execution, the pairs were labeled regarding the malware family, the presence of malware according to VirusTotal verification, the presence of malware considering the difference of sensitive method calls only, and the presence of malware considering the arguments passed to the sensitive methods.

This brings us to the following research questions:

1. How does comparing arguments passed to sensitive API calls impact the results of the MAS approach regarding the detection of Android malware?
2. How does the malware family influence the performance of this technique?

4.1.1 Data Collection

The data for the exploratory analysis was collected by submitting the 1,707 pairs of apps through the modified version of DroidXP. The execution of each pair goes as follows:

1. Instrumentation: in this step, every APK sample is instrumented using DroidFax, by passing the correct parameters to allow the API Tracker module to be executed. It's executed only once per version of each app, given that the results would be the same for every execution.
2. Execution: in this step, DroidXP executes the instrumented APKs on an Android emulator for the specified time, and collects the relevant logs through *logcat*. The emulator used was running at API 28 (Android 9). It's important to remember that DroidXP wipes the emulator data after every execution to avoid interference (such as caches) between the apps.
3. Reporting: this step collects the generated logs and other relevant data generated at the instrumentation step (static analysis results, for example) and groups them for each version of each app, separated by execution.

DroidXP was configured to execute every sample 3 times, for 180 seconds using DroidBot as test case generation tool, and passing the `-p` flag to enable the argument capturing. The choice of DroidBot as test case generation tools is due to previous research showing that it outperforms all other tools when generating tests for malware detection purposes [18].

After DroidXP finished its execution, the results were submitted to the report generator tool to compute the difference of method calls (and arguments passed). If at least one method differ, the app is considered to be malicious.

4.2 Exploratory analysis

After the execution, the results were aggregated and the assertiveness of both approaches (i.e considering only the different sensitive methods called vs considering the methods and the arguments) were compared. The confusion matrices of each approach can be seen on the Table 4.2 and Table 4.3. Each execution have one of the following outcomes, taking VirusTotal as source of truth:

- True Positive (TP), meaning that the app was correctly labeled as malware.
- True Negative (TN), meaning that the app was correctly labeled as benign (not malware).
- False Positive (FP), meaning that the app was wrongly labeled as malware.
- False Negative (FN), meaning that the app was wrongly labeled as benign (not malware).

The approaches were compared by taking into account the *precision*, *recall* and *f-score* (F_1) of data presented at Tables 4.2 and 4.3. Each metric is calculated using the formulas 4.1, 4.2 and 4.3, respectively, and the higher the value the better is the performance of the approach. The calculated scores for both approaches can be seen at Table 4.1.

$$precision = \frac{\mathrm{TP}}{(\mathrm{TP} + \mathrm{FP})} \tag{4.1}$$

$$recall = \frac{\mathrm{TP}}{(\mathrm{TP} + \mathrm{FN})} \tag{4.2}$$

$$F_1 = \frac{2 \cdot precision \cdot recall}{(precision + recall)} \tag{4.3}$$

Table 4.1: Comparison of both versions in terms of precision, recall and F_1 score.

	Precision	**Recall**	F_1
Previous version	0.37	0.30	0.33
New version	0.46	0.48	0.47

By only considering the difference of methods called between each version of each app, the MAS have labeled 403 apps as malware, from which 151 (37.47%) were correct (TP). The other 1,304 apps weren't labeled as malware, from which 965 (74%) were correct (TN). This brings us to 0.33 F_1 score.

When taking into account the methods and the arguments (i.e., an app is considered to be a malware if there's a difference on the sensitive methods called **or** there's a difference

Table 4.2: Results of the MAS malware detection that only considers difference of sensitive methods called

		reference	
		negative	positive
predicted	negative	965	339
	positive	252	151

Table 4.3: Results of the MAS malware detection that considers the difference of sensitive methods called and the arguments passed for them

		reference	
		negative	positive
predicted	negative	940	254
	positive	277	236

on the arguments passed to sensitive methods), the MAS labeled a total of 513 apps as malware, from which 236 (46%) were correct (TP). The remaining 1,194 apps weren't labeled as malware, from which 940 (78.73%) were considered to be correct (TN). This results on 0.47 F_1 score, an overall improvement of 14% if compared to the previous approach.

This answers our first research question:

1. How does comparing arguments passed to sensitive API calls impact the results of the MAS approach regarding the detection of Android malware?

   The analysis show that there's a performance improvement, specially regarding the true-positive dimension with 85 additional cases detected when taking into account the difference of arguments.

The results presented above take the whole dataset of 1,707 apps into account, and did not consider the internal functionality of each malware family. The proposed approach may have different results depending on how each type of malware acts on the devices. For that reason, a new analysis was made grouping the results based on the malware family, in order to investigate if the comparison of arguments passed to sensitive methods can improve the performance of detection of certain types of malware. This new arrangement can be seen on Table 4.4, where the 490 malicious apps are classified based on the malware family, and the detection results of both approaches (only considering the difference of sensitive methods, and considering the difference of both the methods and the arguments).

Considering the families with more than five samples, it considerably improved (20% or more) the detection performance of the *smsreg*, *dowgin*, *appad* and *cauly* families. There was also some improvement on the detection of the *revmob* and *gappusin* malwares, to

a lesser extent. However, some other malware families weren't able to be detected even with the new proposed approach, which is the case for the *kuguo* family.

This result may be due to internal functionality of each malware type: for example, the *dowgin* family is known to be a type of adware [26], a malware that displays unwanted advertisements [27] and don't necessarily makes use of sensitive capabilities of the device other than internet access. If the original version of the app also displays advertisements, the malicious version may redirect the ad revenue (by changing an API key) and it wouldn't be necessary to introduce any new sensitive method calls. Therefore, if that was the case, this malicious app would not be labeled as malware without taking the arguments of methods into account.

With that in mind, it's possible to answer the second research question:

2. How does the malware family influence the performance of this technique?

   The acquired results indicates that there's a big influence of the malware family on the performance of this technique. The detection improves the results of the previous technique by a larger margin on some families, such as *smsreg* that improved 69.23%, while having less impact on some other families such as *revmob*, with 6.45% improvement or *kuguo* with 0%.

Table 4.4: Results of the MAS malware detection with and without argument catching, per malware family

Malware family	**Samples**	**Previous accuracy**	**New accuracy**
gappusin	295	9.49%	22.71%
revmob	31	90.32%	96.77%
dowgin	25	52.00%	72.00%
airpush	21	100.00%	100.00%
leadbolt	13	100.00%	100.00%
smsreg	13	15.38%	84.62%
cauly	7	0.00%	85.71%
kuguo	7	85.71%	85.71%
adwo	6	100.00%	100.00%
domob	6	100.00%	100.00%
appad	6	0.00%	66.67%
youmi	6	50.00%	66.67%
torjok	4	0.00%	50.00%
plankton	3	100.00%	100.00%
shixot	3	0.00%	100.00%
viser	3	100.00%	100.00%
wooboo	3	100.00%	100.00%
dnotua	3	66.67%	66.67%
droidkungfu	3	66.67%	66.67%
boogr	2	50.00%	100.00%
douwan	2	0.00%	100.00%
pushad	2	100.00%	100.00%
smspay	2	100.00%	100.00%
stopsms	2	0.00%	100.00%
leadb	2	0.00%	50.00%
smalihook	2	50.00%	50.00%
antilvl	1	0.00%	100.00%
appax	1	0.00%	100.00%
appflood	1	100.00%	100.00%
appsgeyser	1	100.00%	100.00%
autosms	1	0.00%	100.00%
dianle	1	100.00%	100.00%
elfan	1	100.00%	100.00%
flurry	1	100.00%	100.00%
ginmaster	1	0.00%	100.00%
inmobi	1	0.00%	100.00%
lzla	1	0.00%	100.00%
madad	1	0.00%	100.00%
pircob	1	0.00%	100.00%
uapush	1	100.00%	100.00%
basebridge	1	0.00%	0.00%
odpa	1	0.00%	0.00%
ramnit	1	0.00%	0.00%
uten	1	0.00%	0.00%

Chapter 5

Conclusions

This study shows that there's a lot of room for improvement regarding the malware detection capabilities of the mining sandboxes technique. Our main goal of improving its accuracy by adding a new criteria for malware detection was successfully achieved even though the overall improvement is modest and highly dependant on the malware family that's being analyzed.

The study was performed on a much larger dataset of 1,707 app pairs, if compared to previous studies that were often limited on 100 or less samples. It is noticeable that the malware detection performance of the mining sandboxes is also compromised when analyzing a larger dataset, even with the improvements proposed in this paper.

Future work might explore ways to improve the argument detection, since the method calls might be executed in many ways on the programming languages used by the Android platform, such as reflection or callbacks that weren't handled on the version of the tool implemented for this study, and might also explore a different set of sensitive API calls for the analysis.

The Mining Android Sandboxes approach has been proven to be a useful tool for malware detection on the Android ecosystem with its scalability and automation capabilities, but a lot of research is still necessary in order to turn it into a reliable tool for protecting app marketplaces, devices, and final users from malicious actors.

References

[1] Jung, Jin Hyuk, Ju Young Kim, Hyeong Chan Lee, and Jeong Hyun Yi: *Repackaging Attack on Android Banking Applications and Its Countermeasures*. Wireless Personal Communications, 73(4):1421–1437, December 2013, ISSN 1572-834X. `https://doi.org/10.1007/s11277-013-1258-x`. vii, 4

[2] Costa, Francisco Handrick da, Ismael Medeiros, Pedro Costa, Thales Menezes, Marcos Vinícius, Rodrigo Bonifácio, and Edna Dias Canedo: *DroidXP: A Benchmark for Supporting the Research on Mining Android Sandboxes*. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 143–148, 2020. vii, 2, 6, 7

[3] Turner, Ash: *How Many Android Users Are There? Global and US Statistics (2023)*. `https://www.bankmycell.com/blog/how-many-android-users-are-there`, visited on 2023-07-05. 1

[4] Turner, Ash: *How Many Apps In Google Play Store? (Jul 2023)*. `https://www.bankmycell.com/blog/number-of-google-play-store-apps/`, visited on 2023-07-05. 1

[5] Truong, Hien Thi Thu, Eemil Lagerspetz, Petteri Nurmi, Adam J. Oliner, Sasu Tarkoma, N. Asokan, and Sourav Bhattacharya: *The Company You Keep: Mobile Malware Infection Rates and Inexpensive Risk Indicators*. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, pages 39–50, New York, NY, USA, 2014. Association for Computing Machinery, ISBN 978-1-4503-2744-2. `https://doi.org/10.1145/2566486.2568046`, event-place: Seoul, Korea. 1

[6] Jamrozik, Konrad, Philipp von Styp-Rekowsky, and Andreas Zeller: *Mining Sandboxes*. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 37–48, New York, NY, USA, 2016. Association for Computing Machinery, ISBN 978-1-4503-3900-1. `https://doi.org/10.1145/2884781.2884782`, event-place: Austin, Texas. 1, 5

[7] Bao, Lingfeng, Tien Duy B. Le, and David Lo: *Mining sandboxes: Are we there yet?* In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 445–455, 2018. 1, 2, 5, 6, 16

[8] Costa, Francisco Handrick da, Ismael Medeiros, Thales Menezes, João Victor da Silva, Ingrid Lorraine da Silva, Rodrigo Bonifácio, Krishna Narasimhan, and Márcio

Ribeiro: *Exploring the use of static and dynamic analysis to improve the performance of the mining sandbox approach for android malware identification.* Journal of Systems and Software, 183:111092, 2022, ISSN 0164-1212. https://www.sciencedirect.com/science/article/pii/S0164121221001898. 1, 2, 6, 7, 16

[9] Le, Tien Duy B., Lingfeng Bao, David Lo, Debin Gao, and Li Li: *Towards Mining Comprehensive Android Sandboxes.* In *2018 23rd International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 51–60, 2018. 1, 2

[10] *Platform architecture*, May 2023. https://developer.android.com/guide/platform, visited on 2023-06-09. 3

[11] *Application Sandbox*, October 2022. https://source.android.com/docs/security/app-sandbox, visited on 2023-06-09. 3

[12] *Permissions on Android*, May 2023. https://developer.android.com/guide/topics/permissions/overview, visited on 2023-06-09. 3

[13] Skoudis, Ed and Lenny Zeltser: *Malware: Fighting Malicious Code.* Prentice Hall PTR, USA, 2003, ISBN 0-13-101405-6. 4

[14] Zhou, Yajin and Xuxian Jiang: *Dissecting Android Malware: Characterization and Evolution.* In *2012 IEEE Symposium on Security and Privacy*, pages 95–109, 2012. 4, 5

[15] Felt, Adrienne Porter, Erika Chin, Steve Hanna, Dawn Song, and David Wagner: *Android Permissions Demystified.* In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 627–638, New York, NY, USA, 2011. Association for Computing Machinery, ISBN 978-1-4503-0948-6. https://doi.org/10.1145/2046707.2046779, event-place: Chicago, Illinois, USA. 5

[16] Felt, Adrienne Porter, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner: *Android Permissions: User Attention, Comprehension, and Behavior.* In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, New York, NY, USA, 2012. Association for Computing Machinery, ISBN 978-1-4503-1532-6. https://doi.org/10.1145/2335356.2335360, event-place: Washington, D.C. 5

[17] Hamilton, James and Sebastian Danicic: *An Evaluation of Current Java Bytecode Decompilers.* In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 129–136, 2009. 5

[18] Li, Yuanchun, Ziyue Yang, Yao Guo, and Xiangqun Chen: *DroidBot: a lightweight UI-Guided test input generator for android.* In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 23–26, 2017. 6, 17

[19] Cai, Haipeng: *Understanding Application Behaviours for Android Security: A Systematic Characterization.* Technical report, Department of Computer Science, Virginia Polytechnic Institute & State University, June 2016. https://vtechworks.lib.vt.edu/bitstream/handle/10919/71678/cairyder_techreport.pdf, visited on 2023-06-11. 6

[20] Vallée-Rai, Raja, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan: *Soot - a Java Bytecode Optimization Framework*. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, page 13. IBM Press, 1999. Place: Mississauga, Ontario, Canada. 6

[21] *Soot*. `https://soot-oss.github.io/soot/`, visited on 2023-06-11. 6

[22] *Logcat command-line tool*, April 2023. `https://developer.android.com/tools/logcat`, visited on 2023-06-09. 7, 14

[23] Allix, Kevin, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon: *AndroZoo: Collecting Millions of Android Apps for the Research Community*. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 468–471, New York, NY, USA, 2016. Association for Computing Machinery, ISBN 978-1-4503-4186-8. `https://doi.org/10.1145/2901739.2903508`, event-place: Austin, Texas. 16

[24] *About us - VirusTotal*. `https://support.virustotal.com/hc/en-us/categories/360000160117-About-us`, visited on 2023-06-15. 16

[25] Khanmohammadi, Kobra, Neda Ebrahimi, Abdelwahab Hamou-Lhadj, and Raphaël Khoury: *Empirical study of android repackaged applications*. Empirical Software Engineering, 24(6):3587–3629, December 2019, ISSN 1573-7616. `https://doi.org/10.1007/s10664-019-09760-3`. 16

[26] *Android/Dowgin threat description*. `https://www.f-secure.com/sw-desc/adware-android-dowgin-online.shtml`. 20

[27] *What is Adware? – Definition and Explanation*. `https://www.kaspersky.com/resource-center/threats/adware`. 20