Ares: Triggering Payload of Evasive Android Malware

Luciano Bello
IBM Research
Yorktown Heights, New York, USA
luciano.bello@ibm.com

Marco Pistoia
IBM Research
Yorktown Heights, New York, USA
pistoia@us.ibm.com

ABSTRACT

With the emergence of mobile application markets, there has been a dramatic increase in mobile malware. Mobile platform providers are constantly creating and refining their malware-detection techniques, including static analysis and behavioral monitoring. The goal of malware writers is to hide the malware payload from those analyzers. In parallel, security analysts want to quickly detect if any software is malware in order to prevent harm to users. This confrontation is pushing malware writers to develop new evasion techniques that prevent their malware from being detected or making analysis harder.

This paper describes Ares, a system built on top of an existing behavioral analysis, based on static information-flow analysis, binary instrumentation, and multiexecution analysis, to detect and bypass many common evasive techniques used by mobile malware. Additionally, this paper presents our implementation of Ares, and shows that, when run against real-world software, Ares is able to reveal previously unknown malicious components. We also developed a test suite for evasion detection techniques: Evadroid, which we have made fully available to other researchers.

ACM Reference Format:
Luciano Bello and Marco Pistoia. 2018. Ares: Triggering Payload of Evasive Android Malware. In *MOBILESoft '18: MOBILESoft '18: 5th IEEE/ACM International Conference on Mobile Software Engineering and Systems , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3197231.3197239

1 INTRODUCTION

Detecting malware in the mobile ecosystem is crucial and challenging at the same time. Platform vendors need to protect their users, especially in the context of open app markets, in which apps are provided by third parties. Much of the success of a mobile platform depends on the quality and quantity of third party apps offered in its market. It is in the interest of a vendor to protect its market from malicious apps, usually referred to as *malware*. For this reason, both Google Play and the Apple App Store require that any app distributed to consumers undergo a series of checks—including code inspection, static analysis, sandboxing and testing—in order to detect any security threat. Despite these efforts, attackers have found multiple ways to bypass the security checks performed by the platform vendors (see, e.g., [31] and [22] for the Apple App Store and Google Play, respectively).

MOBILESoft '18, May 27–28, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5712-8/18/05…$15.00
https://doi.org/10.1145/3197231.3197239

In general, malware is said to be *evasive* if it implements techniques that avoid the execution of malicious behavior—the *payload*—under certain circumstances in order to bypass detection or to make the analysis of the malware harder. One of the techniques that malware writers use to evade malware detection is *environment fingerprinting* [21, 23, 30]: evasive malware does not behave maliciously if, based on characteristics of the running environment, it detects that is being monitored.

Evasive techniques have already been observed in the wild for years in the x86 platforms (traditional personal computers, usually running the Microsoft Windows operating system). A large body of research has been dedicated to combating them [3, 11, 12, 16, 18]. As for the mobile setting, in 2011 Zhou and Jiang identified and analyzed AnserverBot—one of the most sophisticated malwares infecting Android devices [40]. By using dynamic code loading, data obfuscation, self-signature verification and encryption, this bot program was particularly good at evading analysis and detection. According to Zhou and Jiang, "*the combination of these techniques significantly raises the bar for reverse engineering analysis.*" Since then, the presence of evasive malicious code should be accounted for when attempting to detect malware.

From the attacker's perspective, writing evasive malware is not just necessary to enter the app market, but also a way to increase the time window in which the attack is effective. Executing the payload immediately might, for example, expose the Command and Control infrastructure (C&C) or the server where personal information is leaked. Uncovering the attacker's infrastructure allows firewalling and might lead to the attacking organization.

In this paper, we present a system that extends an existing behavioral dynamic analysis to improve its chances for triggering the payload of evasive Android malware. The system uses information-flow analysis to detect possible evasion points. In order to force the execution of otherwise-non-taken branches, it instruments multiple copies of the malware. The different alternative copies of the app flip different combination branches, which are executed in a given monitored environment. This environment creates footprint logs of various executions that are used to feed back into the system. To test the system, we implemented Ares, an instantiation of the suggested approach with a simple behavioral analysis that compares execution traces in order to expose the payload. We tested Ares with real-world malware that is known to be evasive to explore its practicability. In addition, we created a set of tests to validate our approach and make it comparable to future alternatives.

In summary, this work makes the following contributions:

A system to trigger the payload in evasive mobile malware. Given an existing dynamic behavioral analysis of mobile apps, the system presented in this papers extends it with the goal of exposing hidden behavior without involving complicated and expensive reverse-engineering processes.

An implementation and evaluation of the system. In order to evaluate our system, we implemented it by developing and combining the necessary components to analyze, instrument, run,

and monitor Android application packages. We called this implementation Ares and we made it fully available for the sake of reproducibility. We evaluated the presented approach with Ares on 10 well-known real evasive malware families. In 9 of the cases, the payload was exposed totally or partially. The evaluation shows that the approach is feasible and practical.

The Evadroid test suite. As part of the development of the Ares framework, we created 22 apps that exhibit evasive behavior. Those minimal apps implement evasive techniques and have been made part of a test suite that is public to use and can be extended by others.

The rest of this paper is organized as follows: Section 2 presents a description of the system including a discussion on optimizations and efficiency improvements. Section 3 illustrates the implementation details of Ares and its evaluation against our test suite, Evadroid, as well as the real-world malware. We discuss some known limitations in Section 4. Section 5 presents related work. Finally, Section 6 concludes the paper with a short discussion of further work.

2 DESCRIPTION OF THE SYSTEM

Consider a hypothetical evasive malware that connects to a C&C for receiving commands as part of a botnet. Keeping the C&C hostname hidden from security analysts is crucial in order to increase the effectiveness of the malware. Some malware hide the hostname by enciphering the server address or obfuscating the code that performs the connection. In addition, in order to avoid giving away the hostname during behavioral analysis, malware might only try to connect when `android/telephony/TelephonyManager.getDeviceId()` returns something different from `0000000000000`, since it is well-known that this device ID is used by many emulators.

On the other side, the goal of a malware analyst is to trigger the payload of evasive malware in a controlled environment in order to learn more about it. The evasive payload is the malicious or unexpected behavior (in the example, the connection to the C&C) that is only executed under certain conditions (in the example, when the device ID is not `0000000000000`), presumably for evading purposes.

The goal of the system proposed by this paper is to help the analyzer to trigger the payload of possibly evasive malware. For that reason, we start by characterizing evasion in malware. The most commonly evasive code follows the pattern in Figure 1.

```
1: fp ← FingerPrinting();
2: if not fp == "reference value" then
3:     (malicious behavior)
4: end if
5: (innocuous behavior)
```

Figure 1: Evasive Pattern

The execution of the malicious behavior depends on the result of FingerPrinting() in line 1. Broadly speaking, this function can be any mean to sense the environment. It can be a constant fetch (like in the case of the device ID) as well as the date, the battery status, or information on the movement sensors, among others. In general, we call this kind of sensing call *fingerprinting source*, or FS for short. In this large meaning, an FS returns a value that gives some information about the conditions under which the malware is running and it can be used for evading purposes.

At line 2, the FS is checked, and a decision branch is opened based on it. A priori, it is not possible to be sure that the value returned by the FS call is used for evading, so we call the branching point *Evasion Point Candidate* (EPC). In this example, the true-side branch executes the malicious behavior, called *payload*, at line 3.

2.1 General Description

The objective is to expose the malware payload by executing it in a monitored environment. For that, we split the general approach into 3 stages:

***Stage 1** - Finding EPCs.* As previously defined, finding an EPC is a data-flow dependency problem usually tackled by *information-flow analyses* [26]. A typical information-flow analysis uses the concepts of *sources* and *sinks*. The analysis detects if there is data originated in sources that is consumed by any sink. In the particular case of evasion scenarios, FS elements are sources and the branching conditions as sinks. That is, if a condition used in a branch depends on data from one or more FSs, we consider that point an EPC. This approach was already explored by previous work in the context of JavaScript malware [14] with its own set of challenges. For this stage, and taking into account the specificities of our Android scenario, we use a static information-flow analyzer for Java bytecode to find EPCs.

Given a malware candidate and a list of FSs, the output of this first stage the set of EPCs. Some evasive payloads can be nested into more than a single EPC. Even further, there is no guarantee that any of these EPCs are used for hiding a payload. Further stages are in charge of determining that.

***Stage 2** - Forcing the Untaken Branch.* This stage takes the list of EPCs from **Stage 1** and the original malware candidate as input. The EPCs are located and instrumented in the malware candidate to force the execution of the false side of the branch. In the case of Java bytecode, we interleave stack operation to flip the result of the conditional instruction. In addition, we interleave a checkpoint in each EPC. This is, we log a mark when an EPC is visited at run time.

The result of each instrumentation is a set of new binaries, each of them is an almost-identical copy of the original malware candidate, where the decision of a certain combination of EPCs is flipped and the originally non-taken branch is executed at run time. The combinatorial strategy is a key element to discover a payload located in nested blocks controlled by several FSs. An increase in the amount of FS considered in **Stage 1** results in more EPCs, which leads to more combinations. Since the amount of new binaries to consider for the next stage directly depends on the amount of EPCs and their combinations, it is critical to combine them strategically, as we discuss further in Subsection 2.3.1.

***Stage 3** - Execution of the Instrumented Malware Candidate.* The multiple binaries from the previous stage are executed in a monitored environment for behavioral analysis. An example of such an environment is the Google Bouncer, an infrastructure where apps are executed before entering in the Google Play market to be analyzed and to detect malicious or forbidden behavior.

In principle, there are no restrictions on the nature of the environment. The result is independent of the hardware, provided that the malware candidate is able to fully run on it. The reason behind this is that, if the malware candidate uses a particular feature of the environment to evade, then the function to extract this feature should be considered an FS. Therefore, the system is, at least in theory, fully agnostic on the running environment.

The only requirement to the environment is the access to the checkpoint logs mentioned in **STAGE 2**. This information is used for creating the EPC flipping combination and should be accessible by the system. In other words, in order to decide combinations of EPCs to flip, it is necessary to know which EPCs have been visited in certain executions.

Besides this requirement, the behavioral analysis is standard and requires no modification. As usual, it runs the malware candidate with environment and user-simulated stimulus with the goal of detecting malicious behavior.

2.2 An Ad-hoc Running Environment

The system runs on top of an existing dynamic environment, such as Google Bouncer, with access to the checkpoints logs. In order to test the approach and since most of the state-of-the-art behavior analyses are not available as open-source tools [23], we have to extend the system with an ad-hoc running environment.

For the purposes of this paper, we will consider a monitored environment that generates execution traces of low-level sensitive API calls, such as opening a network socket, sending an SMS, loading code dynamically, and accessing the file system. This environment is a modified Android OS that logs when certain sensitive functions are called, generating a trace log. The log entry includes the arguments and the call stack as this information can help us to characterize the payload. In this way, and in addition to executing a malware candidate in this environment, all the variations created by the system are executed and their trace logs are compared to expose new execution paths.

Our ad-hoc behavioral analysis installs the instrumented applications generated in **STAGE 2** in the monitored Android environment and stimulates the execution using Monkey [7]. After that, it uninstalls the app and extracts the trace log generated. Once all the call traces are collected we compare them with each other to detect extra executions that might be the expression of the payload. The logs are aligned and the differences in the executions are manually analyzed using comparison tools. The block in the trace log where the payload is executed will be referred to as *payload evidence*.

In our experience, the most effective comparison takes all the instrumented executions against the *vanilla execution*—i.e., the execution of the malware candidate without any instrumentation. It is also possible to analyze single executions by searching for payload elements, such as open ports, that occur only in certain instrumented execution.

Running **STAGE 3** under this setting encounters several issues that are discussed in detail in Subsection 4.1. However, a payload detection mechanism is needed to run the system. It is important to notice that such mechanism is independent of the evasion circumventing approach that is proposed.

2.3 Optimizations and Enhancements

The full system can be explained following the algorithm in Figure 3, which has its graphical representation in Figure 2.

Given a malware candidate *APK* in line 1, the system runs an information-flow analysis to find the set of EPCs (line 2), as described in **STAGE 1**. In line 3, a set of logs L is initialized with a single element: the log trace of the vanilla execution, produced by the execution of the malware candidate without any modification in the monitored environment. The last initialization step is in line 4, when a worklist S is created with possible combinations of EPCs to flip. Each combination $s \in S$ is a set of zero or more EPCs to flip. Further, we process each s element of the worklist S in a for-loop at line 5.

The function INSTRUMENT() in line 6 performs **STAGE 2**. It creates a new version of the malware candidate where the EPCs in s are flipped. The function unpacks the original APK, instruments the Java bytecode in it, repacks it, and signs it. The instrumentation searches for each EPC to flip and negates the condition to force the execution of the non-taken branch. It also inserts a checkpoint that logs when an EPC is visited so that, when the application is run, there will be an indication of the reachable EPCs in the log.

```
1:  procedure MAIN(APK)
2:      EPC ← DEPENDENCYANALYSIS(APK, spec);
3:      L ← {EXEC(APK)};
4:      S ← INITS(EPC, APK);
5:      for all s ∈ S do
6:          APK_s ← INSTRUMENT(APK, s, EPC);
7:          log_s ← EXEC(APK_s);
8:          L ← L ∪ {log_s};
9:          S ← ADD(S, log_s, s);
10:     end for
11:     PayloadEvidence ← ANALYZE(L);
12:     return PayloadEvidence;
13: end procedure
```

Figure 3: Core Algorithm

STAGE 3 starts at line 7, when each instrumented APK is executed in the monitored environment and the trace logs are accumulated in the set of logs L (line 8). The function ADD() in line 9 models the feedback loop that is explained further in the next subsection.

Lastly, the function ANALYZE() in line 11 fully encapsulates the way that the behavioral analysis evaluates the maliciousness of the considered APK. In the case of our ad-hoc analysis, it takes the set of trace logs generated by the monitored executions and compares them. Its objective is to find the *Payload Evidence* by searching for unexpected or malicious actions, such as remote connections or personal information access.

Following, we enumerate some of the design decisions to optimize and enhance the payload detection. Starting with the feedback loop from Figure 2, we also tackle the problem of interprocedural dependencies and a mitigation of evasion using sleep calls.

2.3.1 Efficient Path Exploration. Triggering the payload of evasive malware is a path coverage issue. Sometimes called *predicate coverage*, this requires the combination of each logical condition to execute all the possible program traces. This kind of attempt can easily grow into a combinatorial explosion.

Inspired by DD-path graphs [8], we decided to simplify the execution paths as a complete binary decision tree with EPCs as nodes, left edges for not flipped, and right edges for flipped. In order to determine which EPC can be found in a particular execution path, we use the checkpoints inserted by INSTRUMENT() and backtracking. These checkpoints simply log when an EPC is visited during the execution.

```
1:  procedure INITS(EPC, APK)
2:      APK_{} ← INSTRUMENT(APK, {}, EPC);
3:      log_{} ← EXEC(APK_{});
4:      return VISITEDEPCS(log_{});
5:  end procedure
```

The *log analysis* step from Figure 2 is implemented by VISITEDEPCS(-), which searches for the checkpoint marks in an execution log and returns the set of those EPCs in the trace. During the worklist initialization, the instrumentation does not flip any EPC (as shown by the fact that the second argument is the empty set in line 2), but injects the log calls to detect that an EPC was reachable. Each of the EPCs found will later be individually flipped, potentially leading to new EPCs.

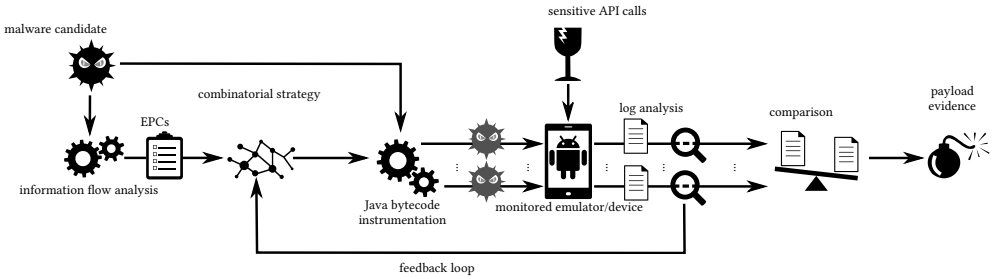

Figure 2: General Workflow of the Full Ares System

The Add() function extends the worklist at the end of every iteration with new EPCs that have been discovered in each execution.

```
1: procedure Add(S, log_s, s)
2:     V ← visitedEPC(log_s);
3:     nS ← V \ S;
4:     for all ns ∈ nS do
5:         S ← S ∪ {s ∪ {ns}};
6:     end for
7:     return S;
8: end procedure
```

The feedback loop from Figure 2 is implemented by $\text{Add}(S, log_s, s)$ function, which extends the set S based on the new explored path. After each execution of an instrumented APK in which the s EPCs were forced to flip, the trace log log_s is analyzed looking for the EPCs in that trace (line 2). Each of those newly discovered EPCs (denoted as nS in line 3) is combined with s to extend S (line 5). In this way, a backtracking algorithm for path exploration is implemented.

As an example, consider the situation where function DependencyAnalysis() finds 5 EPCs, denoted as ①, ②, ③, ④ and ⑤, respectively. The successive extensions of S are detailed in Figure 4. Figure 4a represents the state after the initial execution in line 3 of Init(), in which EPCs ① and ② were visited. Let ◇ denote the end of the execution, whether this happens as a consequence of timeout or a program crash, which might take place for the reasons explained further in Section 4.4. In Figure 4b, EPC ② is flipped, denoted as ②̄, and two new EPCs are discovered: ③ and ⑤. These are combined with ② and added to S, in line 5 of the algorithm. We removed the explored flipping combinations from S for clarity. The rest of the steps are as expected, until each element in S is exhausted.

2.3.2 Interprocedural Analysis. A common problem of static information-flow analyzers is the extensive over-approximation that takes place when modeling interprocedural settings. A sound interprocedural analysis requires a precise call graph and heap representation. In the information-flow context, this creates over-tainted data – i.e. over-approximations that create false positives. More false positives translate into more EPCs whose flip do not lead to more chances to find the payload snippet.

In our scenario, we decided to simplify the interprocedural analysis by dynamically extending the list of FS functions. When an FS flows to a return statement instead of a branching condition, we add the method in which this happens to the list of FS functions, and we rerun the analysis. This way, if in other parts of the code there is a branching depending on a method that might return a tainted value, we capture that branching as an EPCs.

2.3.3 Evasion through Sleep Calls. Some malware might try to evade detection by delaying the payload waiting for the behavioral analysis to stop the execution. For example, it is well-known that the Google Bouncer analyzes an app by running it for five minutes [22]. It is then trivial to hide the malicious behavior by just waiting for that time before activating the payload.

We decided to mitigate this form of evasion by nullifying the most common form of delay evasion: *sleep calls*, which are primitive functions to turn the program inactive for a certain period of time.

Our instrumentation (function Instrument, line 6 in Figure 3) searches for instances of sleep calls and replaces the argument with a zero. This approach, called *sleep acceleration*, is also used by previous work that implements countermeasures against evasion techniques [19].

Because sleep calls also govern timeouts and retry operation attempts, they cannot be fully ignored at the operating-system level. When the delay depends on information from APIs, like in the case of triggering the payload in a particular date, we consider them as FSs. Other forms of delays, such as using stalling code, cannot be handled with sleep acceleration and are discussed in Section 4.

3 INSTANTIATION OF THE SYSTEM: ARES

While the ideas presented in the paper are independent of any particular implementation, in order to test the approach we implemented Ares: a toolchain to help unveiling the payload of evasive Android malware.

3.1 Implementation and Evaluation Settings

Ares consists of an APK coder/decoder, an information-flow-security analyzer, a Java bytecode instrumenter, a modified Android operating system running in an emulator or in a device, and a tool to compare files, as well as *glue* scripting code to orchestrate all the components. The full implementation of Ares is publicly available in the URLs listed in Section 7. We used APKTool 2.0.3 [32] for unpacking and packing APKs. A handful of Python 2 and Bash scripts were used to process the files and automate tasks such as installing and removing the packages in the environment.

For **Stage 1** we wrote an information-flow tracker using WALA 1.3.6 [36], which supports Java bytecode analysis. Our tracker takes the FSs and statically analyzes the Java bytecode, searching for the branching opcodes that depend on those FSs. The decisions on which FSs to select is a trade-off: a very generic source (such as a constructor) creates many EPCs (and most likely, many APKs), while a very specific source may increase the precision, but might be bypassed by obfuscation. This specific selection is based on the existing literature of evasive Android malware and it can be modified or extended in other scenarios.

We wrote a bytecode instrumenter using ASM 5.0.3 [34] for **Stage 2**. When one of the opcodes is encountered, we pop the

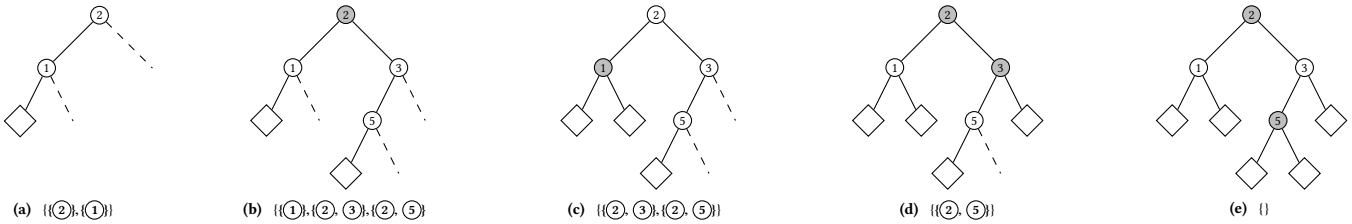

Figure 4: Backtracking to explore EPC flipping combinations.

arguments from the stack and push new arguments that force the execution of the other side of the branching. The instrumenter also nullifies the sleep method calls for sleep acceleration.

The installation, execution, and stimulation of an app in the monitored environment for **STAGE 3** is done by Bash scripts calling ADB and Monkey [7]. We installed a custom Android Open Source Project 5.1.51 on an emulator and, for performance reasons, on a Nexus device. The customization of the Android code is the insertion of log calls `java/lang/System.log` in sensitive functions, such as ClassLoaders and network connectors.

The log analysis for the feedback loop and the combinatorial strategy is implemented in a Python 2 script. For the log comparison we used icdiff 1.8.1 [10] in combination with ad-hoc scripts to find key operations, like extracting URLs from network-connection attempts.

During the development of ARES, and inspired by previous work and known malware, we built a test suite: EVADROID. Then, to study the viability of our approach in real scenarios, we evaluated ARES against both EVADROID and known malware.

The summary of these evaluations can be seen in Tables 1 and 2. Column *# of EPCs* reports the number of EPCs found by the dependency analysis for each case. Consequently, Column *unreachable EPCs* corresponds to the number of EPCs that were not executed by any of the traces. This gives an idea of the code that was not covered. Column *sleep calls* indicates the number of nullified sleep calls. The number of generated APKs is reported in Column *# of APKs*. This includes also APKs with combinations of flips, following the strategy explained in Section 2.3.1. Following, we report the number of APKs whose executions ended in crashes. Given that the tests in EVADROID are very simple, no crashes were detected and we removed that column from Table 1 for clarity.

When the payload is totally or partially detected by ARES after the execution of all the generated EPCs, we marked them with ✓ and ✓ respectively. When this happened, we also reported in which iteration the payload was detected and how many simultaneous flips were needed to trigger it. Since in EVADROID every test has the same payload, we excluded this column in Table 1. We used ✗ to denote a situation in which the payload was not detected.

The following sections explain the most interesting cases of the evaluation. All the raw data from where these results are extracted can be found following the links in Section 7.

3.2 EVADROID 1.0 Test Suite Evaluation

Table 1 summarizes the results of ARES against EVADROID 1.0. We extracted evasive techniques from existing malware and from previous work and reimplemented as minimal Android apps. In all the tests, the payload is exactly the same: send one or more premium SMSs. We designed these tests with the purpose of highlighting the benefits and defects of ARES, as well as making it comparable to future approaches.

In most cases, only two APKs are generated, since there is a single evasion point. Most of the unreachable EPCs were found as a result of over-approximations in the dependency analysis, which taint other branching points in the SDK library included in every app. This can be noticed by the fact that the number of EPCs is always similar. Because most of the boiler-plate code is never executed, the same situation applies to the number of unreachable EPCs.

The test *constants2* needed more than one flip to trigger the payload. As the rest of *constant** tests, this case makes use of multiple FSs, but in the following arrangement. This creates a set a nested conditionals at bytecode level.

```
if (Build.MODEL.contains("sdk") &&
    Build.MODEL.contains("emulator")&&
    Build.PRODUCT.startsWith("sdk")&&
    Build.BRAND.equals(Build.DEVICE)&&
    Build.FINGERPRINT.startsWith("generic")
   ) { payload }
```

The tests *sleep* and *postDelay* make use of sleep calls, and no flipping was necessary to trigger the payload, just the nullification of these calls. In addition, notice that a single APK was generated, since no EPC was visited during the execution. In the same direction, *longAction* and *constantDLC* do not exercise any EPC. The first one, simulates the sleep call with a time consuming I/O action. In our setting, this action took around 20 minutes and delayed the execution of the payload over the limit of the monitored execution. Because this test does not make use of FSs or sleep calls, the payload was not detected by ARES. The *constantDLC* has the evasion mechanism in dynamically loaded code and is also not detectable. The situation of *qemuFingerprinting* also loads code dynamically (native code in this case). However, the evasion is detected since the native code is used just for fingerprinting, while the evasion itself is performed in the main class. By considering `System.load Library` a FS the evasion is detected.

The last situation that was not captured by ARES is *divById*, which has the following structure:

```
try { int invId = 1 / toInt(getDeviceId());
    payload
} catch (ArithmeticException e) { harmless behavior }
```

The test triggers a division-by-zero exception when `getDev iceId` is composed only by zeros (common device ID in many emulators), creating a branch without conditional opcodes. Since our setting does not account for this form of implicit flows, the dependency is not detected and the evasion point is missed. These three cases where detection was not possible are discussed in detail in Section 4.

In total, 86% of the test cases were detected. Each test generated an average of 2.8 APKs, and most of them triggered their payload in the second execution or earlier. On average, each test contained

Table 1: Results for the Evadroid 1.0 test suite

Name	# of EPCs	unreachable EPCs	sleep calls	# of APKs	detected?	in iteration	simultaneous flips
accelH	123	122	0	2	✓	2	1
adbEnable	123	122	0	2	✓	2	1
adbPortDetector	126	122	0	5	✓	2	1
atNight	124	122	0	3	✓	2	1
batteryCharging	126	125	1	2	✓	2	1
batteryFull	126	125	1	2	✓	2	1
batteryStatus	123	122	0	2	✓	2	1
constantCalls1	126	122	0	5	✓	3	1
constantCalls2	125	122	0	4	✓	3	1
constants1	129	122	0	8	✓	5	1
constants2	127	122	0	6	✓	6	4
constantsDLC	129	129	0	1	✗		
divById	123	122	0	2	✗		
getIpAddress	125	124	0	2	✓	2	1
installedApps	123	122	0	2	✓	2	1
longAction	122	122	0	1	✗		
postDelayed	122	122	1	1	✓	1	0
procNetTcp	128	125	0	4	✓	2	1
qemuFingerprinting	123	122	0	2	✓	2	1
signatureVerification	123	122	0	2	✓	2	1
sleep	122	122	1	1	✓	1	0
uptime	123	122	0	2	✓	2	1
Average	124.6	122.8	0.18	2.8	86%	2.4	1.1

124.6 EPCs, but only 1.8 could be exercised. On one hand, this could be an advantage, because less reachable EPCs mean less APKs to instrument and test. On the other hand, it puts out a relative low coverage and that might translate into an unavailability of finding the payload in real malware. Hence, there is a need for validating these results with malicious apps from the wild.

3.3 Ares against Real Malware

We ran Ares on 10 instances from different malware families. We took known malware that showed some for of evasive mechanism in hand-made reports. The objective is test the approach against real-case scenarios. We focused on diversity, as we considered old and new instances, as well as simple and very complex families. The newest instance is from the Android.Spy.277.origin family, which was first seen by VirusTotal in October 2016. In contrast, we considered very classic malware, such as AnserverBot and BaseBridge.

The results of our analysis is summarized in Table 2. Ares exposed the payload in 7 over the 10 considered cases and, in addition, exposed a partial payload in 2 other cases. With an average of 75.5 APKs generated for each candidate, we argue the approach is feasible. Specially because the payload was found much quicker (in iteration 31.9, in average). The footnotes are the SHA256 sum of the used instances, for a quick reference in the VirusTotal website [33].

3.3.1 Connection to C&C Servers. Many forms of malware have the goal of handling the control of the infected device to an external entity. The attacker can control many devices from a centralized location, known as C&C. Hiding the hostname of this C&C is key for the attack to last longer. If the C&C host is known, it can be firewalled or closed by a judicial order.

Malware such as AnserverBot, BaseBridge, and JSmsHider are examples that include several features to hide the C&C host such as

Table 2: Results for the malware families

Name	# of EPCs	unreachable EPCs	sleep calls	# of APKs	# of crashes	detected?	payload	in iteration	simultaneous flips
Android.Spy	1964	1590	60	376	19	✓	APK download	109	3
AnserverBot	86	67	2	41	1	✓	remote host connection	34	3
Banker-IR	43	5	3	39	0[&]	✓	change default SMS app	20	2
BaseBridge	142	108	12	35	8	✓	remote host connection	14	3
Deng.KJF	646	576	8	71	70[&]	✓	remote host connection	20	1
DroidCoupon	179	84	10	96	6	✓	unpack exploit	68	2
Dropdialer	2	2	0	1	0	✗			
Fakemart	10	3	2	8	1	✓	premium SMSing	2	1
JSmsHider	89	64	0	26	2	✓	remote host connection	4	1
SmsReg	950	874	32	62	30	✓	APK download	6	2
Average/Total	411.1	337.3	12.9	75.5	13.7	7		31.9	2

aggressive code obfuscation and encryption. [17, 29, 40] We used Ares to analyze an instance of each of these families. A simple string search of URLs in these kind of malware would not reveal the C&C hostname or its request parameters. Because Ares works on top of a behavioral analysis, it did not have to attack the obfuscation or the encryption to force the instance to call home.

For the **AnserverBot**[1] case, the instance we analyzed connects to a C&C hosted at `b4.cookier.co.cc:8080` in the 34th iteration of Ares. The request sends device id information and waits for commands. The host, which at the time of writing this paper resolves to a Korean IP, is up and running but without the port 8080 opened. As Android.Spy.277.origin, this malware family also checks the package signature to ensure that the app was not manipulated. Ares automatically circumvented this protection because the function to get the package signature is considered an FS.

Ares shows that the analyzed instance of **BaseBridge**[2], as AnserverBot, connects to `b3.cookier.co.cc:8080` C&C server in the 14th iteration. It sends phone identification information in the HTTP GET request. According to the NetQin Mobile *"upon activation, the malware would activate three malicious services – AdSmsService, BridgeProvider and PhoneService – to communicate with a control server"*[17]. Ares was able to trigger that activation and was able to expose the C&C host without any human interaction.

Lastly, the analyzed instance of **JSmsHider**[3] communicates to the C&C using DES encryption [29]. This behavior can be observed in the 4th iteration of Ares, when the malware communicates to the C&C `http://svr.xmstsv.com/Notice/` after accessing the `javax.crypto` package. Because our monitoring reports on the arguments of cryptographic calls, Ares was able to expose the DES key used for the encrypted communication, which is `"[B@192f53"` in this particular instance. The malware initially sends information about the infected device.

3.3.2 Access to Remote APKs. A special case of the previously explained remote connection is the access to a remote APK. Malware usually uses this technique as a way to bypass the market restrictions. Ares discovered two instances that connected to a remote server from where they tried to download new packages to install in the victim system. These instances belong to the families Android.Spy.277.origin and SmsReg.

[1] `10032dc19f609ed8b7577a5620c87a1bd9605ef0b12654e5983982cd640c0d89`
[2] `58781d1e86b8ea935c6ae7145b0a46e70e92d10e39f02404dc5bfab6e4d1bded`
[3] `7c940a56ec8522b6c84607f822dde8f5fb48480312583f3d1fd81e9af02f6e6b`

The analyzed **Android.Spy.277.origin**[4] downloaded a remote APK from `http://uhay.vn/ngach/fixpolac_50030.apk`, during the iteration 109. This file was still available at the moment of writing this paper and its maliciousness was confirmed by VirusTotal[5]. A string analysis of the instance does not expose the hostname nor the APK name.

The remote request is triggered in a separate thread and runs in the background. In order to expose this behavior 3 EPCs had been simultaneously flipped. Part of the evasion is a signature checked that the malware performs to detect modifications.

The access to the remote malicious APK matches the manual analysis that have been reporting about this family. For example, a Check Point analysis mentions an attempt to *"download the malicious APK called polacin.io to the device remotely through the C&C server"*[15]. The main package name of the APK accessed by Ares is `com.polac.ingen_dft30`, supporting the idea that we are, indeed, referring to the same malicious APK.

In addition, our instrumentation also reports that personal information (such as model, OS version, and the like) is leaked to `api.pingstart.com:17209` directly, without any evasion mechanisms. Although this leak is arguably malicious, since PingStart is a well-known ad platform.

In the case of **SmsReg**[6], a decompilation of this instance exposed the strings `android.51mrp.com:8077` and `dl.elevensky.net`. During the execution of the 8th iteration, Ares showed that the first host is used to call home. This host was up and running at the moment of writing this paper, although blacklisted by Google. Ares allowed further conclusions by showing, in the 6th iteration, that the malware fetches two APKs the second location mentioned. Because we performed an analysis that exhausts the combination of reachable EPCs, we were able to discover a total of 7 APKs that were downloaded from `dl.elevensky.net` At the moment of this writing all these APKs were still available for download.

An analysis from VirusTotal indicates that all these 7 APKs are malicious at different levels. Two of them where unknown by VirtusTotal when submitted by us, in September 2016. Since the analyzed instance of SmsReg was first submitted to VirusTotal in May 2015, there is a possibility that these APKs have never been discovered before Ares exposed them.

3.3.3 Other Payloads. The DroidCoupon family was described in detail by Jiang [9]. As Jiang describes, it tries to escalate root privileges by running an exploit known as *RageAgainstTheCage*, which exploits a vulnerability called *adb setuid exhaustion*. In the instances of **DroidCoupon**[7] analyzed by Ares, the exploit is obfuscated and packed. Ares triggered and showed the unpacking process of the exploit into the file `/data/data/cn.buding.coupon/files/rageagainstthecage` during the 68th iteration of our method. We were available to confirm the exploit.

In addition to exposing the unpacking process of the exploit, this execution is also the only one loading the native library `libandroidterm.so`[8]. The library is included in the original APK and it is considered to be malicious by VirusTotal.

We also analyzed with Ares a particular instance of the **Fakemart**[9] family created by Petsas et al. to test one of their anti-analysis techniques [23]. It is a modification of a simpler instance of *Fakemart* that sends premium SMSs which use no evasion. The modification from Petsas et al. wraps the payload in an evasion technique that detects the execution on QEMU-based emulators. They called the technique *xFlowH* and it is explained in detail in Section 2.3 of [23].

When Petsas et al. tested this instance of Fakemart with xFlowH with 12 well-known dynamic analysis tools, none of them could detect the payload. However, Ares detected the payload by considering the calls to native code as FS. That is, if a decision is made based on the result of a native code call, that decision is a possible evasion point. In the second iteration, by flipping the single point where the emulation environment is checked, the payload was exposed.

3.3.4 Partial or No Payload Exposed. As Ares is neither sound nor complete (explained in the Section 4), we encountered three instances where the perform was the expected one: Banker-IR, Deng.KJF, and Dropdialer. In the first two cases, we considered that Ares detected part of the payload (notated as ✓[3] in Table 2). This is, situations where we know, for example, from previous reports that their payloads include several stages and some of them were not triggered.

The instance of **Banker-IR**[10] that we took was previously analyzed in detail [1]. The malware has typical evasive strategies to avoid running the payload in emulated environments. According to its report, the malware have a two-staged payload: it attempts to receive device administrator rights as to contact its C&C server at the same time. While Ares was able to trigger the administration right request to the user in the 20th iteration, we did not observe any connection to the C&C.

Also we consider a partial detection the situation with the analyzed instance of **Deng.KJF**[11].

During the 20th interation, the instance opens the browser for accessing `m.74443.com`, with sensitive information (such as the location and the device ID) in the GET parameters. This websites with adult content does not appear in a string analysis of the sample, but it is embedded in many detected malware files, according to VirusTotal[12] and should be considered dangerous. Since there are many strings matching IP forms in the decompilation of the instance, we were expecting to see connection to them. However, most of executions finish in crashes and, therefore, we have to assume those connections remain hidden and the payload was not triggered.

Lastly, let us consider the case of **Dropdialer**[13] which, according to the reports, in June 2012 evaded the Google Bouncer by fingerprinting the environment and was downloadable for 2 week in the Google Play store [20]. Based on this, we took an instance of it with high expectations of capturing the payload. However, Ares was not able to find enough EPCs to trigger any malicious behavior. This could be the consequence of incompleteness in the FS list or in the static dependency analysis.

[4] `82afc2da4ea0404a3a1fa9756b3c85853a63fd26d657fa1d9fcd6c0955e2a84e`
[5] `ced1c2e4924ac5905e1a0a613b0699b2b02c1e95f0a67bd86aa5b342fa9a0be1`
[6] `1c802423b1c87d637fb9f6d0027ce4cde0be43462ca3bfe8159c3541d27b6da5`
[7] `94112b350d0feceff0a788fb042706cb623a55b559ab4697cb10ca6200ea7714`
[8] `2b5c883a5678768e831006b8f4fb04ae35fe5ba70157f8accf0d8dd5d2c27ce8`
[9] `5e1cf973a37a9460ddfdaee46a318ad7a59c2f99dd02d14362a0c15f8b6d4adc`
[10] `4ab8f26e8aaee3de12b04b7a86be9ee349672e228b52e5b90dcd63cf7b564e34`
[11] `8c6ba45515ccf0e48db450aa2aa6bab23b3ef06302cd1c2abaced4254d355a30`
[12] https://www.virustotal.com/en/domain/m.74443.com/information/
[13] `0ca20e6fa0b57583dff39e0ab25d43c14beb410afac5569a9e5aaadabd2f1932`

3.4 Performance

The static analysis and instrumentation time is negligible in our single desktop settings where Ares was executed. In general, the system multiplies the analysis time of a regular behavioral analysis by the amount of APKs generated to find the payload. Since the architecture is trivially parallelizable –the behavioral analysis can be simultaneously run in several elements of the worklist– we consider the approach feasible to be implemented on existing analyses.

4 LIMITATIONS AND THREATS TO VALIDITY

Although this approach towards forcing the execution of mobile malware payload is incomplete and unsound, it works in many scenarios and moves the arms race to the next level. In the process of developing Ares, we identified challenges for further steps, including some possible solutions.

4.1 The Considered Running Environments

Even though the general approach of this paper is independent of the running environment that defines and detects the malicious payload, we had to put an environment and a detection criteria in place in order to test it. Our focus was not to compete with well-tested existing behavioral analyses, but to extend them. As a consequence, there is a lot of room for improvement in this aspect to make Ares a fully comparable and functional system. Better forms of dynamic analysis have been studied in the past, with proper input stimuli and sandboxing [21, 23]. There are two main aspects to consider in the presented running environment: the trace comparability and the lack of good input stimulus.

We detected the payload by comparing the trace logs generated by the execution of the instrumented malware candidate. The execution reproducibility is crucial to facilitate the comparison of the trace logs. However, an exact execution condition is hard to achieve. There are many sources of non-determinism and that makes trace log aligning intricate. The problem of identifying and eliminating sources of non-determinism for differential analysis of network traces was attacked by Continella et al. [2]. In our case, the notion of trace is more generic than the mere network activity and their work cannot be reused in this context.

For this reason, we performed the trace comparison mostly manually, with help from some scripts and comparison tools such as `icdiff` [10], with the intention of focusing on reproducible systems as part of future work. More complex ways to align execution traces had been studied before (e.g. [11]) and it might be beneficial to explore alternatives on this issue.

With respect to proper stimulus, running an application requires user-like input (tapping in buttons, voice interactions, etc.) that is hard to simulate automatically [6]. Our environment interacts with the app by randomly tapping the screen and pressing the buttons (such as speaker volume) while triggering all the registered intends. We stimulate the app very basically using Monkey [7] but it might not be enough. A respectable amount of literature about this problem and possible solutions is discussed in Subsection 5.2.

4.2 Static Analysis Incompleteness

In Ares we use WALA for the dependency analysis, which is fully static. That means, *Dynamically Loaded Code* (DLC) is not analyzed, making situations like the *constantsDLC* test from Evadroid impossible to detect, since the evading branching is in DLC. Notice that our approach could fully detect if just the payload lives in DLC, since this code would run on the monitored environment. Similarly, we were able to detect *qemuFingerprinting* because the `System.loadLibrary` method is considered a FP while the EPC is part of the analyzed code.

Another source of incompleteness comes from *reflection*, which is hard to handle statically. A possible solution would be to move the dependency analysis to a dynamic approach. Similarly, handling DLC is challenging [28, 39]. In principle, it would be possible to detect a DLC situation, analyze it, and instrumented, all on the fly at running time. *Dila* is a WALA effort to extend the analysis to DLC [35]. However, this would complicate the system substantially and would move the problem to dynamically loaded native code, or *JNI*. This will make the system dependent on the running environment, which conflicts explicitly with our design decision.

For scalability reasons, the static dependency analysis we used does not consider implicit flows. Implicit flows occur when the data dependency is through the control flow of the program, in opposition to the direct assign. In the following two examples, there is data flowing to y from the result of the FP() call. On the left-hand side, that flow is explicit (y depends on x and x depends on FP()) while on the right-hand side, the dependency is implict y value depends on a condition, that depends on FP().

```
1: x ← FP();
2: y ← x + 5;
```

```
1: y ← 0
2: if FP() == 1 then
3:     y ← 6;
4: end if
```

Given that in our system the branching points are sinks, implicit flow in this basic sense is covered out-of-the-box. However, there are other forms of implicit flow. A notable case are *exceptions*, like in the *divById* test from Evadroid. Exceptions create branches that are very coarse to capture statically [4]. A dependency analysis with support for implicit flow created by exceptions would create a huge amount of EPCs, since almost every statement is prone to raise exceptions.

4.3 Delays by Stalling Code

When a malware is tested dynamically, the execution needs to be finished at some point. In the particular case of our monitored execution, we run the instrumented malware candidate for some seconds. By considering functions depending on the date or hour as FS, we can capture some forms of time bombs. Additionally, we mitigate some forms of delay evasion by nullifying sleep calls (see Subsection 2.3.3).

However, an attacker could create any time-expensive operation (sometime known as *stalling code*) to delay the execution of the payload. These operations tend to be inefficient and noisy and could be detected by some heuristic, but they are threats that needs to be considered. We discuss the existing related work in the area in Subsection 5.3.

4.4 Imprecise Modeling and Breaking the Invariant

The use of decision flow graphs (see Figure 4) is an additional potential source of imprecision, since it ignores the effect of loops. This shortcuts also ignores cross-edges and joint points, leading to unwanted situations. For example code blocks might escape forced execution in cross-edges or while loops might never finish.

A similar problem might occur in certain conditional branches. Forcing to take a branch by negating the condition in the guard breaks the block invariant. For example, if the condition checks

the existence of a file, the true side of the branch might correctly assume that the file exists and tries to read it. This means that our instrumentation might lead to a lot of crashes. The Column *# of crashes* on Table 2 reports on the amount of crashes when all the APKs are tested. The general assumption is that the payload does not depend on the condition, which most probably depends on the FS exclusively. But it is important to notice that the crash produced by a broken invariant might also be used for evading or just affecting the functionality of the application, making it impossible to execute it. For example, a broken invariant can make an app to loop endlessly.

Some possible solutions to this problem have been studied in the x86 setting and are discussed in Subsection 5.1. These approaches, instead of changing the branching guard, search for the possible output of the FS method that takes the other branching side. This avoids the problem crashing problem, but adds complexity and might reduce coverage when the constrains are hard to solve.

5 RELATED WORK

To the best of our knowledge, the most related work to the presented approach if from Rasthofer et al. [24]. While using an alike forced execution and a combination of static and dynamic analyses, their goal is slightly different: extract *values of interest* from the malware. These values are constants used for calling sensitive methods, such as phone numbers, URLs and the like. Instead of creating multiple versions of the malware as in our approach, they create a single *reduced* application which is the minimal code that composes the value of interest. Since the semantic change necessary for creating the reduced application is much more invasive than in our case, a behavioral analysis performed on the result is further off from the original software to analyses.

On the other hand, their work circumventing reflection could definitely improve the results of our system. However, the closed nature of their work makes a step in this direction difficult. In terms of their limitations originated by the static analysis and the forced execution of untaken branches, their description is remarkably comparable to ours.

A recent work from Fratantonio et al.[3] introduced *TriggerScope*, a system to detect logic bombs in Android apps. Under their definition, a logic bomb is a piece of code that triggers under very specific circumstances. This definition fits perfectly with our description of evasive behavior. TriggerScope uses static analysis to detect these triggers and focuses on targeted malware (such as part of an advanced persistent threat or state-sponsored attack). However, they cover three specific types of triggers which are based on time, location, and SMSs arrival. The scalability of the system needs to be studied when considering more types of triggers. ARES includes a significantly larger list of triggers (see the extended version of this paper for a comprehensive list of FSs, Section 7) and, because it works on top of a dynamic analysis, provides evidence of the evasive behavior.

In the remainder of this section we report the most related work on the vast corpus on evasive malware using path exploration for x86 platforms. Then, we move to the mobile setting for reporting the main work on evasive malware and its connection with the simulation of user stimuli. Lastly, we broadly discuss evasion techniques based on delays.

5.1 Similar Ideas on x86

Kirat and Vigna studied evasive Windows x86 malware with BareCloud [12]. Similar to us, their malware candidates are run on a monitored sandboxed environments and compared. By comparing the execution of malware under simulated and bare-metal environments, they are able to extract the payload. Since they monitor all the system calls and not just sensitive functions, their log comparison is much more complicated. For this reason, they use bioinformatics algorithms to align and extract the payload [11]. Their notion of evasion is only based on the difference of running in full emulated and real environments. As a consequence, other form of evasion like based on sleep calls, on date/time, or on the interaction with the environment (such as the accelerometer measurement in our case) are ignored.

Moser et al. also attempt to trigger evasive Windows x86 malware, but with a much broader notion of evasion than Kirat and Vigna, including time bombs depending on date, the existence of a file or directory, the Internet connection, or even when C&C commands are received. They perform a path exploration of the decision tree with a similar strategy to ours, but they manipulate the FS instead of the EPC. When an EPC is reached, a snapshot of the execution is taken and the execution continues. Then, the input is manipulated in a bottom-up fashion to force the execution of the untaken branch and performs a path exploration. Their implementation maintains the state of the execution and is at much lower level than ours, including an *inverse memory mapping* to trace back to the FS, and constrain solvers to decide alternative returned values. There is also extra complexity with network connection and file system synchronization leads to a very intricate monitored environment. After all, many paths might end up unexplored. On the positive side, their system can report on which was the evasive condition that the malware was using.

The main strategy of DVasion, proposed by Gilboy [5], is very similar to the one from Moser et al. but, without state maintenance. DVasion is probably the closer implementation for x86 to our approach in mobile. Released in 2016, DVasion manipulates the `eip` register to force the execution path in the instrumented binary and uses dynamic taint analysis to find the FS-EPC dependencies, instead of static analysis like us. Besides those differences, our approaches are similar (one for x86, there other for Android) and our limitations are consistent to his.

5.2 Stimulus-based Malware Detection

As part of our **STAGE 3**, we run the instrumented malware candidate in a monitored environment. Sometimes, the candidate needs user interaction to fully execute. There are many attempts to improve the monitored emulated environments that simulate user interaction to improve code coverage. Following, a summery of them.

Reina et al. introduced CooperDroid in 2013, an analysis build on top of QEMU with the capability to stimulate behaivor on the analysed malware [25]. CopperDroid allows to perform behavioral analysis, deducing from very low level syscall sequences to high level actions such as *sending an SMS*. The stimulation consists of simulating user level action to discover additional malware behavior, which can be considered the payload. CopperDroid tries to avoid evasion by simulating a bare-metal a much as possible, which might be insufficient in case of targeted evasion based on CopperDroid fingerprinting evasion or time-bombs.

Similarly, Andrubis is also focused on recording low level calls while stimulating the malware candidate with user simulation [37]. Their dynamic analysis is aided by a static information-extraction pre-stage. The dynamic part is performed in an emulated environment, just like in CopperDroid, and taint-tracks private information to monitors the behavior of the potentially harmful app.

CopperDroid and Andrubis complement our approach in **Stage 3**: the execution on the monitored environment. As mentioned in Subsection 4.1, the stimulation we provide the app with might not be enough. Most probably, a more complex stimulus would increase the chances to trigger payloads and code coverage. Sadly, both Andrubis and CopperDroid web services were off-line at the moment of writing this work. They are also not open source projects, making impossible to integrate them into our approach.

Most importantly, both attempts might no be enough to fully attack evasion, as Vidas and Christin studied [30]. They focused on techniques to evade analysis based on hardware and software components, as well as performance and behavior of the environment. For this purpose, they tested and fingerprinted different malware detectors running on emulators such as the mentioned CopperDroid and Andrubis among others. Beyond fingerprinting by simple API methods like `getDeviceId` they also explored the use of sensors and settings for that goal.

In the same way, Petsas et al. [23] suggested a taxonomy on evasion techniques while testing them in multiple emulator-based analyzers, including CopperDroid and Andrubis. They are going a step further by profiling the underlying hypervisor. They suggested many countermeasures to increase the realism in the emulator.

Given that our approach tries to trigger branches depending on profiling and fingerprinting methods, we can handle these situations to a much longer extend than a single stimulated execution. Our FS-to-EPC technique allows us to model many forms of sensor based evasion (e.g. a payload triggered when it is shaken over certain threshold), time-based condition (e.g. payload that is triggered in a certain date), and hypervisor characteristics. From both studies (Petsas et al. and Vidas and Christin) we took many ideas to extend Evadroid and showed Ares can be handled these situations.

5.3 Delay-based Evasion

Out of the mobile world delay-based evasion is very common and well studied. Singh et al. distinguish between two possibilities: sleep calls and time triggers [27]. In x86 sandboxing, it is common to monitor for sleep calls and accelerate them, similar to our technique explained in Subsection 2.3.3. For the second kind, our approach can handle time triggers directly, by considering date and time retriever methods as FS (for details in which are these methods. In the mobile setting, Yang et al. included time-based evasion as part of the device capability to read external-environment state [38]. Based on their observations, we included these time-based evasions in Evadroid and showed that they are handled well by Ares.

A third sometimes-ignored possibility besides sleep calls and time triggers is stalling code. As discussed in Subsection 4, stalling code is challenging from the theoretical point of view. Kolbitsch et al. studied instances of x86 malware using stalling code in detail [13]. They suggest heuristics to detect when the malware got stuck in a loop for long and forcing the exit. While, to the best of our knowledge, there is no report of mobile malware using stalling code to delay the execution of its payload, it is a matter of time. The mobile platforms are more limited and easier to monitor than x86 architectures and, probably, stalling code cannot be as stealth as in x86. But in any case, future approaches should start considering mitigation techniques for that.

6 CONCLUSION AND FUTURE WORK

This paper describes a system for helping mobile malware analyzers to trigger malware payloads on behavioral dynamic monitors. The approach is easy to automatize and can be used with existing infrastructure for dynamic analysis, such as Google Bouncer. We implemented and tested the system against our novel Evadroid malware test suite, and 10 real-world malware families, showing that the approach is feasible and practical. In 9 of the malware cases, the system was able to partially or completely exposed the payload, without any need for code deobfuscation or reverse engineering.

The work we presented here tries to step ahead in the race between mobile malware writers and analysts. Current behavioral analyses can be easily extended to increase the chances to find the payload at the cost of running several instrumented versions of the candidate without any extra human interaction. Even though the experimental results that we obtained by running Ares against Evadroid and the real-world malware are very promising, it is easy to predict that mobile malware will turn more complex and even more evasive in the future, most probably following the path already seen in the x86 platforms. Many of the accounted limitations had been studied for x86 in the past years and we are planning to port them to the mobile setting in further work.

7 AVAILABILITY

We released the source code of every developed tool for this work, including Ares (`https://ibm.biz/AresSystem`) and Evadroid (`https://ibm.biz/Evadroid`), as well as the raw results of our evaluations. We also provided the patch file to the Android source code. The information-flow analysis includes the WALA library binaries and the intrumenter includes the ASM libraries. The freely available tools, such as icdiff, APKTools, and Bash and Python interpreters, as well as the Android source code are not included.

Also in `https://ibm.biz/AresSystem`, we published an extended version of this paper with appendices where we list: the functions that have been nullified, the fingerprinting sources and sinks used in WALA, details of each of the Evadroid cases, and a full description of the findings for each of the malware samples considered in this paper.

ACKNOWLEDGMENT

The authors would like to thank Omer Tripp and Pietro Ferrara for their valuable contributions to early stages of this work.

Special thanks to the authors of [23]: Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis, for sharing their evasive code with us.

REFERENCES

[1] Avast: Android banker trojan preys on credit card information (2016), https://blog.avast.com/android-banker-trojan-preys-on-credit-card-information/

[2] Continella, A., Fratantonio, Y., Lindorfer, M., Puccetti, A., Zand, A., Kruegel, C., Vigna, G.: Obfuscation-resilient privacy leak detection for mobile apps through differential analysis. In: Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS). pp. 1–16 (2017)

[3] Fratantonio, Y., Bianchi, A., Robertson, W., Kirda, E., Kruegel, C., Vigna, G.: Triggerscope: Towards detecting logic bombs in android applications. In: 2016 IEEE Symposium on Security and Privacy (SP). pp. 377–396 (May 2016)

[4] Genaim, S., Spoto, F.: Information Flow Analysis for Java Bytecode, pp. 346–362. Springer Berlin Heidelberg, Berlin, Heidelberg (2005), http://dx.doi.org/10.1007/978-3-540-30579-$8_2$3
[5] Gilboy, M.R.: Fighting Evasive Malware with DVasion. Master's thesis, University of Maryland, College Park, USA (2016)
[6] Gomez, L., Neamtiu, I., Azim, T., Millstein, T.: Reran: Timing-and touch-sensitive record and replay for android. In: Software Engineering (ICSE), 2013 35th International Conference on. pp. 72–81. IEEE (2013)
[7] Google: UI/application exerciser monkey | android studio. https://developer.android.com/studio/test/monkey.html (2017), accessed: 2017-06-08
[8] Huang, J.C.: An approach to program testing. ACM Comput. Surv. 7(3), 113–128 (Sep 1975), http://doi.acm.org/10.1145/356651.356652
[9] Jiang, X.: Security alert: New Android malware – DroidCoupon – found in alternative Android markets (2017), https://www.csc.ncsu.edu/faculty/jiang/DroidCoupon/
[10] Kaufman, J.: icdiff - side-by-side highlighted command line diffs. http://www.jefftk.com/icdiff, accessed: 2017-06-08
[11] Kirat, D., Vigna, G.: Malgene: Automatic extraction of malware analysis evasion signature. In: Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security. pp. 769–780. CCS '15, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2810103.2813642
[12] Kirat, D., Vigna, G., Kruegel, C.: Barecloud: Bare-metal analysis-based evasive malware detection. In: Proceedings of the 23rd USENIX Conference on Security Symposium. pp. 287–301. SEC'14, USENIX Association, Berkeley, CA, USA (2014), http://dl.acm.org/citation.cfm?id=2671225.2671244
[13] Kolbitsch, C., Kirda, E., Kruegel, C.: The power of procrastination: Detection and mitigation of execution-stalling malicious code. In: Proceedings of the 18th ACM Conference on Computer and Communications Security. pp. 285–296. CCS '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/2046707.2046740
[14] Kolbitsch, C., Livshits, B., Zorn, B., Seifert, C.: Rozzle: De-cloaking internet malware. In: Security and Privacy (SP), 2012 IEEE Symposium on. pp. 443–457. IEEE (2012)
[15] Koriat, O.: In the wild: Google can't close the door on android malware | check point blog. http://blog.checkpoint.com/2016/04/22/in-the-wild-google-cant-close-the-door-on-android-malware/ (2016)
[16] Kruegel, C.: Full system emulation: Achieving successful automated dynamic analysis of evasive malware
[17] Mobile, N.: Fee-deduction malware targeting android devices spotted in the wild. https://www.netqin.com/en/security/newsinfo$_4$202$_1$.html (2011), accessed: 2017-06-08
[18] Moser, A., Kruegel, C., Kirda, E.: Exploring multiple execution paths for malware analysis. In: Proceedings of the 2007 IEEE Symposium on Security and Privacy. pp. 231–245. SP '07, IEEE Computer Society, Washington, DC, USA (2007), http://dx.doi.org/10.1109/SP.2007.17
[19] Mourad, H.: Sleeping your way out of the sandbox. SANS Institute InfoSec Reading Room (2015)
[20] Musil, S.: Malware went undiscovered for weeks on google play (2012), https://www.cnet.com/news/malware-went-undiscovered-for-weeks-on-google-play/
[21] Mutti, S., Fratantonio, Y., Bianchi, A., Invernizzi, L., Corbetta, J., Kirat, D., Kruegel, C., Vigna, G.: Baredroid: Large-scale analysis of android apps on real devices. In: Proceedings of the 31st Annual Computer Security Applications Conference. pp. 71–80. ACM (2015)
[22] Oberheide, J., Miller, C.: Dissecting the android bouncer. https://duo.com/blog/dissecting-androids-bouncer (2012), accessed: 2017-06-08
[23] Petsas, T., Voyatzis, G., Athanasopoulos, E., Polychronakis, M., Ioannidis, S.: Rage against the virtual machine: Hindering dynamic analysis of Android malware. In: Proceedings of the Seventh European Workshop on System Security. pp. 5:1–5:6. EuroSec '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2592791.2592796
[24] Rasthofer, S., Arzt, S., Miltenberger, M., Bodden, E.: Harvesting runtime values in android applications that feature anti-analysis techniques. In: NDSS (2016)
[25] Reina, A., Fattori, A., Cavallaro, L.: A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In: Proceedings of the 6^{th} European Workshop on System Security (EUROSEC). Prague, Czech Republic (April 2013)
[26] Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on selected areas in communications 21(1), 5–19 (2003)
[27] Singh, A., Bu, Z.: Hot knives through butter: Evading file-based sandboxes. Threat Research Blog (2013)
[28] Smaragdakis, Y., Balatsouras, G., Kastrinis, G., Bravenboer, M.: More Sound Static Handling of Java Reflection, pp. 485–503. Springer International Publishing, Cham (2015), http://dx.doi.org/10.1007/978-3-319-26529-$2_2$6
[29] Strazzere, T.: Security alert: Malware found targeting custom ROMs (jSMSHider). https://blog.lookout.com/blog/2011/06/15/security-alert-malware-found-targeting-custom-roms-jsmshider/ (2017)
[30] Vidas, T., Christin, N.: Evading Android runtime analysis via sandbox detection. In: Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security. pp. 447–458. ASIA CCS '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2590296.2590325
[31] Wang, T., Lu, K., Lu, L., Chung, S., Lee, W.: Jekyll on ios: When benign apps become evil. In: Usenix Security. vol. 13 (2013)
[32] web: APKTool - a tool for reverse engineering android apk files. https://ibotpeaches.github.io/Apktool/, accessed: 2017-06-08
[33] web: VirusTotal - free online virus, malware and url scanner. https://www.virustotal.com/, accessed: 2017-06-08
[34] web: ASM kernel description. http://asm.objectweb.org/ (2017), accessed: 2017-06-08
[35] web: Dila - dynamic load-time instrumentation library for java. http://wala.sourceforge.net/wiki/index.php/GettingStarted:wala.dila (2017), accessed: 2017-06-08
[36] web: WALA - T.J. Watson libraries for analysis. http://wala.sourceforge.net/ (2017), accessed: 2017-06-08
[37] Weichselbaum, L., Neugschwandtner, M., Lindorfer, M., Fratantonio, Y., van der Veen, V., Platzer, C.: Andrubis: Android malware under the magnifying glass. Vienna University of Technology, Tech. Rep. TRISECLAB-0414 1, 5 (2014)
[38] Yang, W., Xiao, X., Andow, B., Li, S., Xie, T., Enck, W.: Appcontext: Differentiating malicious and benign mobile app behaviors using context. In: Proceedings of the 37th International Conference on Software Engineering - Volume 1. pp. 303–313. ICSE '15, IEEE Press, Piscataway, NJ, USA (2015), http://dl.acm.org/citation.cfm?id=2818754.2818793
[39] Zhauniarovich, Y., Ahmad, M., Gadyatskaya, O., Crispo, B., Massacci, F.: 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. pp. 37–48. CODASPY '15, ACM (2015), http://doi.acm.org/10.1145/2699026.2699105
[40] Zhou, Y., Jiang, X.: An analysis of the anserverbot trojan (2011)

A EVADROID 1.0 TEST SUITE DETAILS

We split the tests in categories. As part of the test suite, we also include scripts for handling logs and documentation on expected behavior when the conditions are met or not.

Abnormal Setting

`installedApps`: Fetches the list of installed apps with `getPackageManager` and checks if `com.android.development` is among them. It does not trigger the payload if it is installed.

`uptime`: It triggers the payload only when the application is launched in a system that have been running for more than 4 hours. It calls `uptimeMillis()` for detecting the uptime of the system.

Analysis Detection

`adbEnable`: It uses `Settings.Global.ADB_ENABLED` to detect if Android Debug Bridge (ADB) is enabled and triggers the payload when it is not.

`adbPortDetector`: It parses the file `/proc/net/tcp` to detect if there is an open port between 5555 and 5585 port number. These ports are used by the ADB daemon.

`signatureVerification`: Inspired by AnserverBot [40], checks if the app was repacked by checking the signature.

Emulator Constants Vidas and Christin[30] suggested a set of constants to detect an emulated host environment. The tests `constantCalls1`, `constantCalls2`, `constants1`, and `constants2` check for these constants to avoid the execution of the payload under emulated conditions.

`constantCalls1`: It checks for the following `TelephonyManager` methods: `getDeviceId()`, `getSimSerialNumber()`, and `getSubscriberId()`. They return well-known constants in most of the emulators.

`constantCalls2`: It checks for the following `TelephonyManager` methods: `getLine1Number()` and `getVoiceMailNumber()`. They return well-known constants in most of the emulators.

`constants1`: It checks for `Build.MANUFACTURER`, `Build.SERIAL`, `Build.BOARD`, and `Build.HARDWARE`. These are well-defined constants in most of the emulators.

`constants2`: It checks for `Build.MODEL`, `Build.PRODUCT`, `Build.BRAND`, `Build.DEVICE`, and `Build.FINGERPRINT`. These are well-defined constants in most of the emulators.

constantsDLC: Dynamically loads the `constants1.apk` and runs it. It behaves as `constants1`.

divById: Divides 1 for the sum of the integer representation of each char in `getDeviceId()`. In many emulators, this sum should be 0, and the payload is triggered only in cases where the `ArithmeticException` is not thrown.

qemuFingerprinting: Petsas et al. [23] described a set of heuristics to detect an emulated host environment. One of them is the so-called *xFlowH* heuristic, which checks for constants in QEMU VM. This test runs that heuristic. The native library `libbt_detect2.so` was provided by the authors of the mentioned paper.

Network

getIpAddress: Only triggers the payload when the app is started on a devices with an IP different to 0.0.0.0. The IP is fetched with `getIpAddress()` (from `WifiInfo`).

procNetTcp: One of the heuristics from Petsas [23] is *netH*, which tests for typical emulator network setting in `/proc/net/tcp`.

Sensors Vidas and Christin [30] suggested monitoring changes in the battery to detect an emulated host environment. The tests `batteryCharging`, `batteryFull`, and `batteryStatus` do this.

accelH: Inspired by Petsas et al. [23], it triggers the payload when the device is shaken.

batteryCharging: It uses `BatteryManager.EXTRA_STATUS` to detect when the battery is full and only triggers the payload in such an event.

batteryFull: It uses `BatteryManager.EXTRA_STATUS` to detect if the device is charging. Since, by default, most of the emulators simulated charging devices, it triggers the payload when the status is not charging.

batteryStatus: It detects fluctuation (charging or discharging) in the battery load. Since, in general, emulators do not simulate these fluctuations, it only triggers the payload when they occur.

Time

atNight: Inspired by Yang et al. [38], this test triggers its payload when the app is started between 11pm and 5am.

longAction: This test triggers the payload in a thread (to avoid blocking the UI thread) after reading all the accessible files and coping their content in `/dev/null`. This action is I/O expensive and takes several minutes.

postDelayed: This test triggers the payload in a thread (to avoid blocking the UI thread) after waiting 30 minutes. It uses `postDelayed()`.

sleep: This test triggers the payload in a thread (to avoid blocking the UI thread) after waiting 30 minutes. It uses `sleep`.

B SOURCES FOR WALA

Methods:

```
Landroid/content/Context.getPackageManager()Landroid/content/pm/PackageManager;
Landroid/content/Context.getSharedPreferences(Ljava/lang/String;I)Landroid/content/SharedPreferences;
Landroid/telephony/TelephonyManager.getDeviceId()Ljava/lang/String;
Landroid/telephony/TelephonyManager.getSimSerialNumber()Ljava/lang/String;
Landroid/telephony/TelephonyManager.getSubscriberId()Ljava/lang/String;
Landroid/telephony/TelephonyManager.getCellLocation()Landroid/telephony/CellLocation;
Landroid/telephony/TelephonyManager.getLine1Number()Ljava/lang/String;
Landroid/telephony/TelephonyManager.getCellLocation()Landroid/telephony/CellLocation;
Landroid/telephony/TelephonyManager.getNeighboringCellInfo()Ljava/util/List;
Ljava/lang/System.loadLibrary(Ljava/lang/String;)V
Ljava/lang/System.currentTimeMillis()J
Landroid/os/SystemClock.uptimeMillis()J
Landroid/os/SystemClock.elapsedRealtime()J
Landroid/os/SystemClock.elapsedRealtimeNanos()J
Landroid/location/LocationManager.getLastKnownLocation(Ljava/lang/String;)Landroid/location/Location;
Landroid/location/LocationManager.isProviderEnabled(Ljava/lang/String;)Z
Landroid/location/LocationManager.requestLocationUpdates(Ljava/lang/String;JFLandroid/location/LocationListener;)V
Landroid/content/Intent.getIntExtra(Ljava/lang/String;I)I
Landroid/content/Intent.getIntArrayExtra(Ljava/lang/String;)[I
Landroid/content/Intent.getAction()Ljava/lang/String;
Landroid/content/Intent.getBooleanArrayExtra(Ljava/lang/String;)[Z
Landroid/content/Intent.getBooleanExtra(Ljava/lang/String;Z)Z
Landroid/content/Intent.getBundleExtra(Ljava/lang/String;)Landroid/os/Bundle;
Ljava/lang/System.nanoTime()J
Ljava/util/Calendar.getTime()Ljava/util/Date;
Ljava/util/Calendar.getTimeInMillis()J
Ljava/util/Date.<init>()V
Landroid/net/ConnectivityManager.getActiveNetworkInfo()Landroid/net/NetworkInfo;
Landroid/net/ConnectivityManager.getNetworkInfo()Landroid/net/NetworkInfo;
Landroid/net/ConnectivityManager.getAllNetworkInfo()[Landroid/net/NetworkInfo;
Landroid/net/ConnectivityManager.getAllNetworks()[Landroid/net/NetworkInfo;
Ljava/net/NetworkInterface.getInetAddresses()Ljava/util/Enumeration;
Landroid/net/wifi/WifiInfo.getIpAddress()I
Landroid/net/NetworkInfo.isConnectedOrConnecting()Z
Landroid/net/wifi/WifiManager.isWifiEnabled()Z
Landroid/net/wifi/WifiManager.getConnectionInfo()Landroid/net/wifi/WifiInfo;
Landroid/content/Context.getSystemService(Ljava/lang/String;)Ljava/lang/Object;
```

Fields:

```
Landroid/os/BatteryManager.EXTRA_STATUS:Ljava/lang/String;
Landroid/os/Build.MANUFACTURER:Ljava/lang/String;
Landroid/os/Build.BRAND:Ljava/lang/String;
Landroid/os/Build/VERSION.SDK:Ljava/lang/String;
Landroid/os/Build.HARDWARE:Ljava/lang/String;
Landroid/os/Build.FINGERPRINT:Ljava/lang/String;
Landroid/os/Build.MODEL:Ljava/lang/String;
Landroid/os/Build.PRODUCT:Ljava/lang/String;
Landroid/os/Build.SERIAL:Ljava/lang/String;
Landroid/os/Build.USER:Ljava/lang/String;
Landroid/os/Build.CPU_ABI:Ljava/lang/String;
Landroid/os/Build.CPU_ABI2:Ljava/lang/String;
Landroid/os/Build.BOARD:Ljava/lang/String;
Landroid/os/Build.DEVICE:Ljava/lang/String;
Landroid/os/Build.HOST:Ljava/lang/String;
Landroid/os/Build.ID:Ljava/lang/String;
Landroid/os/Build.DISPLAY:Ljava/lang/String;
Landroid/os/Build.TYPE:Ljava/lang/String;
Landroid/os/Build.TAGS:Ljava/lang/String;
Landroid/content/pm/PackageInfo.signatures:[Landroid/content/pm/Signature;
Landroid/hardware/SensorEvent.values:[F
Ljava/util/Calendar.HOUR_OF_DAY:I
Ljava/util/Calendar.DATE:I
Ljava/util/Calendar.MONTH:I
Ljava/util/Calendar.YEAR:I
Ljava/util/Calendar.DAY_OF_WEEK:I
Ljava/util/Calendar.SECOND:I
Ljava/util/Calendar.MINUTE:I
```

Constants:

```
"/proc/net/tcp"
"adb_enabled"
"debug_app"
"development_settings_enabled"
"device_provisioned"
"http_proxy"
"wait_for_debugger"
"samsung"
"89014103211118510720"
"310260000000000"
```

```
"15552175049"
"15555215554"
"310260"
"android.net.conn.CONNECTIVITY_CHANGE"
"android.intent.action.BOOT_COMPLETED"
"android.provider.Telephony.SMS_RECEIVED"
"android.intent.extra.PHONE_NUMBER"
"android.intent.action.NEW_OUTGOING_CALL"
"android.intent.action.RUN"
"android.intent.action.DELETE"
"android.intent.action.VIEW"
"android.intent.action.PHONE_STATE"
"android.intent.action.PACKAGE_ADDED"
"android.intent.action.PACKAGE_REMOVED"
"android.intent.action.PACKAGE_CHANGED"
"android.intent.action.PACKAGE_REPLACED"
"android.intent.action.PACKAGE_RESTARTED"
"android.intent.action.PACKAGE_INSTALL"
```

C SINKS FOR WALA

Opcode(args)	hex (dec)	Meaning
ifeq(v)	0x99 (153)	$v = 0$
ifne(v)	0x9A (154)	$v \neq 0$
iflt(v)	0x9B (155)	$v < 0$
ifge(v)	0x9C (156)	$v >= 0$
ifgt(v)	0x9D (157)	$v > 0$
ifle(v)	0x9E (158)	$v <= 0$
if_icmpeq(i_1, i_2)	0x9F (159)	$i_1 = i_2$
if_icmpne(i_1, i_2)	0xA0 (160)	$i_1 \neq i_2$
if_icmplt(i_1, i_2)	0xA1 (161)	$i_1 < i_2$
if_icmpge(i_1, i_2)	0xA2 (162)	$i_1 >= i_2$
if_icmpgt(i_1, i_2)	0xA3 (163)	$i_1 > i_2$
if_icmple(i_1, i_2)	0xA4 (164)	$i_1 <= i_2$
if_acmpeq(r_1, r_2)	0xA5 (165)	$r_1 = r_2$
if_acmpne(r_1, r_2)	0xA6 (166)	$r_1 \neq r_2$
ifnull(r)	0xC6 (198)	r = null
ifnonull(r)	0xC7 (199)	$r \neq$ null

D SLEEP FUNCTIONS (METHODS)

```
android/os/Handler.postDelayed(Ljava/lang/Runnable;J)Z
java/lang/Thread.sleep(J)V
android/os/SystemClock.sleep(J)V
java/lang/Object.wait(J)V
java/util/concurrent/TimeUnit.sleep(J)V
```

E TRIGGERING DETAILS IN THE ANALYSED MALWARE

The appendix details the conditions in which the analyzed malware from Section 3.3 trigger the payload.

Android.Spy.277.origin In the iteration 109 the malware downloads a malicious APK from a remote location. This iteration simultaneously flips 3 EPCs: the 59th conditional in `com/sweet/rangermob/b/e.a([Ljava/lang/String;)Ljava/lang/String;`, the 17th in `com/sweet/rangermob/xser/RangerSer$13.a(Lorg/json/JSONObject;)V` and the first one in `com/sweet/rangermob/helper/j.ae(Landroid/content/Context;)I`.

AnserverBot In the iteration 34 the malware connects to a remote C&C host. This iteration simultaneously flips 3 EPCs: the 2nd conditional in `com/sec/android/providers/drm/European.a(Ljava/lang/String;)V`, the 1st in `com/android/view/custom/BaseABroadcastReceiver.onReceive(Landroid/content/Context;Landroid/content/Intent;)V` and the 11th in `com/sec/android/providers/drm/Onion.a(Landroid/content/Context;Landroid/content/Intent;Landroid/content/BroadcastReceiver;Ljava/io/FileDescriptor;Ljava/lang/String;)Z`.

The signature check of the package is done by `com/sec/android/providers/drm/Union.class`, which is invoked from the mentioned `onReceive`.

Banker-IR In the iteration 20 the malware attempts to change the default SMS application. This iteration simultaneously flips 2 EPCs: the 1st conditional in `jgywwv/jvyjsd/sordvd/Activity1.onCreate(Landroid/os/Bundle;)V` and the 1st conditional in `jgywwv/jvyjsd/sordvd/AlarmReceiverSmsMan.onReceive(Landroid/content/Context;Landroid/content/Intent;)V`

BaseBridge In the iteration 14 the malware connects to a remote C&C host. This iteration simultaneously flips 3 EPCs: the 8th conditional in `com/android/battery/a/sx.a()Z` and the first ones in `com/android/battery/a/aq.b(Landroid/content/Context;)Z` and `com/android/battery/BaseBroadcastReceiver.onReceive(Landroid/content/Context;Landroid/content/Intent;)V`.

Deng.KJF In the iteration 20 the malware connects to a remote host and leaks personal information. This iteration flips a single EPC: the 2nd conditional in `com/umeng/analytics/b.a(Landroid/content/Context;Landroid/content/SharedPreferences;)V`

DroidCoupon In the iteration 68 the malware unpacks a exploit. This iteration simultaneously flips 2 EPCs: the 2nd conditional in `cn/buding/coupon/core/loaddata/c.a(Landroid/content/Context;)V` and the 4th in `cn/buding/coupon/core/SystemService.onStart(Landroid/content/Intent;I)V`.

Fakemart In the iteration 2 the malware unpacks a exploit. This iteration flips a single EPC: the 1st conditional in `com/android/blackmarket/BlackMarketAlpha.onCreate(Landroid/os/Bundle;)V`.

JSmsHider In the iteration 4 the malware connects to `svr.xmstsv.com:80`. This iteration flips a single EPC: the 1st conditional in `com/AudioConsole/AudioConsole.Reportresult(Ljava/lang/String;)V`. is flipped (in iteration 4), the connection to is exposed.

In the iteration 18 the malware attempts to connect to `10.0.0.172:80`. This iteration flips a single EPC: the 1st conditional in `com/AudioConsole/SocketHttpRequester.ApachedoPost(Landroid/content/Context;Ljava/lang/String;Ljava/util/Map;Ljava/lang/String;)[B`.

SmsReg A total of 7 APKs are fetch from the host `dl.elevensky.net`, flipping several combinations of EPCs.

In the iteration 6 the malware downloads the first two APKs: `wKhklVVtdKOAVv5IAAFBG_mUhK4571.apk`[14] and `wKhklVfjO2-AGUmHAAcw93GVdqc073.apk`[15]. This iteration flips simultaneously 2 EPCs: the 1st conditional in `com/adr/yykbplayer/aa.isNetworkConnected()Z` and the 3rd in `com/adr/yykbplayer/aa.showTime()V`.

In iteration 8 the malware downloads two new APKs: `wKhklFehVYyAAY6fAACrUW6FC_M627.apk`[16] and `wKhklFfjO1eAIO3FAALKlEzyYks334.apk`[17]. This iteration flips simultaneously 2 EPCs: the 1st conditional in `com/adr/yykbplayer/aa.onCreate(Landroid/os/Bundle;)V` and the 3rd in `com/adr/yykbplayer/aa.showTime()V`.

[14] 9ec72143fc0236bc0308d937f98ec117c525c96430d8050081b60a880a72bbaa
[15] e36567e82cb1a703f083cae744c165b56c90f2d40d067888b3ff9dc44dc2d10d
[16] f116af210ce69116f25c5513ff811affbcf0ae5a977d8e95498e6c2ad479e9b1
[17] f9861e8396c40d3c0edf32e1bc7b03258e5436bbca7e8cf15d0761605ee5f342

In iteration 36 the malware downloads two new APKs: `wKhklFeVx_eAGgbeAADsl9ji_ZY770.apk`[18] and `wKhklFfY5N-AJy5SAARfCO2-O7s446.apk`[19] This iteration flips simultaneously 3 EPCs: the very same two mentioned in the iteration 6 and the 10th conditional in `com/skymobi/pay/newsdk/util/PluginLoader.a(Ljava/lang/String;)Ljava/lang/String`.

In iteration 42, the last APK is download: `wKhklVaDLdCANWhcAASZJ-KJAx0897.apk`[20] This iteration flips simultaneously 3 EPCs: the very same two mentioned in the iteration 6 and the 7th conditional in `com/skymobi/pay/newsdk/util/PluginLoader.a(Ljava/lang/String;)Ljava/lang/String;`.

[18] `c4a8a02e900f4fb066a0e8d4c9e2976c9a0f252729058b2915fdc93eae65af49`
[19] `50a7e5196d113ecb12760aadc200573278fb083732d077667625c6f78f678614`
[20] `45d9ed5bfd5898d2426cfad9947c79fd41db6deb05bb0fb97f00e979d1d10804`