Glassbox: Dynamic Analysis Platform for Malware Android Applications on Real Devices

Paul Irolla and Eric Filiol
Laboratoire de Cryptologie et Virologie Opérationnelles (CVO Lab), École d'Ingénieurs du Monde Numérique (ESIEA), 38 Rue des Docteurs Calmette et Guérin, 53000 Laval, France

Keywords: Dynamic Analysis, Android, Malware Detection, Automatic Testing.

Abstract: *Android* is the most widely used smartphone OS with 82.8% market share in 2015 (IDC, 2015). It is therefore the most widely targeted system by malware authors. Researchers rely on dynamic analysis to extract malware behaviors and often use emulators to do so. However, using emulators lead to new issues. Malware may detect emulation and as a result it does not execute the payload to prevent the analysis. Dealing with virtual device evasion is a never-ending war and comes with a non-negligible computation cost (Lindorfer et al., 2014). To overcome this state of affairs, we propose a system that does not use virtual devices for analysing malware behavior. ***Glassbox*** is a functional prototype for the dynamic analysis of malware applications. It executes applications on real devices in a monitored and controlled environment. It is a fully automated system that installs, tests and extracts features from the application for further analysis. We present the architecture of the platform and we compare it with existing *Android* dynamic analysis platforms. Lastly, we evaluate the capacity of *Glassbox* to trigger application behaviors by measuring the *average coverage of basic blocks* on the ***AndroCoverage*** dataset (AndroCoverage, 2016). We show that it executes on average 13.52% more *basic blocks* than the *Monkey* program.

1 INTRODUCTION

Google reacted to the rise of malware with a dynamic analysis platform, named *Bouncer* (Lockheimer, 2012), that analyzes applications before the release on Google Play. This security model is centralized and acts before the distribution of applications. Whereas this system suffers from limitations — like virtual device evasion — it has helped to reduce the malware invasion by 40% (Lockheimer, 2012).

Android antivirus companies use another centralized security model which acts after the distribution of applications. Because applications have access to restricted resources and permissions, antivirus programs cannot perform their analysis — as it often requires root permissions and extensive resources. Hence, the static analysis is externalized onto the company servers. As a result, it can give quick responses — each application being analyzed just once. This is a shift of security model for the common user toward centralization.

Users have been used to decentralized security model, i.e. personal antivirus. This security model does not allow much room for manoeuvre because any antivirus needs to be quick enough for not bothering the user — otherwise another quicker antivirus will be chosen. Whereas antivirus have implemented heuristic algorithms, they are rather limited by the security model. Hence, the security model shifting is an opportunity for building more complex systems that require more resources to run. It enables security systems to use advanced research techniques like behavioral detection with dynamic analysis, or detection with features similarity from static analysis.

Malware authors made their strategy evolve with the rise of *Bouncer* and other dynamic analysis systems. They have started to hide the payload execution with emulation detection or/and the requirement of a user interaction. For example the reverse of the sample (Dharmdasani, 2014) shows that malware are currently using emulation evasion. Emulators settings can be modified to mimic the appearance of a real device but there are lots of ways of detecting *Android* emulation. Actually the tool *Morpheus* (Jing et al., 2014) proves us this war is already lost as the authors found around 10 000 heuristics to detect *Android* emulation. So trying to modify the emulator to look real is probably a waste of time. In such condition, we

Irolla, P. and Filiol, E.
Glassbox: Dynamic Analysis Platform for Malware Android Applications on Real Devices.
DOI: 10.5220/0006094006100621
In *Proceedings of the 3rd International Conference on Information Systems Security and Privacy (ICISSP 2017)*, pages 610-621
ISBN: 978-989-758-209-7

need to redefine the problematic.

This is why we are presenting *Glassbox*, a dynamic analysis platform for *Android* malware applications on real devices. *Glassbox* is an environment for the controlled execution of applications, where the *Android* OS and the network are monitored and have the capacity to block some actions of the analyzed application. This environment is paired with a program that automates the installation, the testing of applications and the cleaning of the environment afterwards. We called this program *Smart Monkey*, as a reference to *Monkey*, the *Android* tool for generating pseudo-random UI event. The objective of *Glassbox* is to collect features for machine learning algorithm, to classify applications as malware or as benign. In the following sections we will present the related work about dynamic analysis systems, we will expose the architecture of both *Glassbox* and *Smart Monkey*. Finally, we will present the *average coverage of basic blocks* of *Smart Monkey* on the *AndroCoverage Dataset* (AndroCoverage, 2016).

2 RELATED WORK

Such dynamic analysis systems started being designed since 2010 (Bläsing et al., 2010) by the academic research, to circumvent the limitations of static analysis — namely, code morphism and obfuscation. Since that time, many systems have been released. For this study we have built a classification of a part of these systems, presented in Table 1 and Table 2. The classification takes into account three categories: the dynamic features collected by the analysis, the strategies set in order to automate application testing and finally the use of real devices in dynamic analysis systems history. We discuss the results on the following sub-sections.

2.1 Features Analyzed

Since the rise of *Android* dynamic analysis systems, the use of system calls have been the leading approach. System calls are the functions of the kernel space, available to the user space. It gives the capacity to manipulate hard drive files or to control processes. System calls can describe a program behaviors, from a low level perspective. The retrieval of those calls can be achieved in two ways, mainly:

- **Virtual Machine Introspection** — This is a technique available for emulators, which enables the host to monitor the guest. It cannot be detected by the guest since it is out of its reach and it is therefore convenient for security analysis. *Andrubis* (Lindorfer et al., 2014), *CopperDroid* (Tam et al., 2015) and *DroidScope* (Yan and Yin, 2012) take advantage of VMI to retrieve, unseen by the target malware, all systems calls done by the guest *Android* virtual machine.
- **Strace/ptrace** — *Strace* is a Linux utility for debugging processes. It can monitor system calls, signal deliveries and changes of process state. *Strace* use the *ptrace* system call to monitor another process memory and registers. This second method is by far the simplest and the most straightforward one as the only task here is the automation of the *strace* execution. Moreover, it targets directly the system calls of the application we need to. That is why this method has been adopted in most of the literature, namely *Crowdroid* (Burguera et al., 2011), *Maline* (Dimjašević et al., 2016), (Canfora et al., 2015) and (Afonso et al., 2015). We have also chosen to use the *strace* utility for system calls monitoring. Despite the theoretical possibility of a malware to detect that it is being debugged, we found no evidence about this.

System calls seem to give great results for classification. *Maline* reported 96% accuracy rate, and (Canfora et al., 2015) reported 94.9% accuracy rate on unseen applications with syscalls frequencies only. Actually syscalls capture low level behaviors of both Java code and native code.

The second most collected feature is taint tracking information as it reveals data leakage. It works by the instrumentation of the *Dalvik VM* interpreter. The information we do not want to leak is called a *source*. Some *source* of personal data are tainted, like the phone number or the contacts list. Each time a tainted *source* or value is used in a method call, the *DVM* interpreter taints the returned value. With this simple mechanism, we can observe the propagation of the tainted information regardless of its transformations. A function that enable to transmit an information outside of the system is called a *sink*, like network requests or SMS. If a tainted value is used in a *sink*, it means data *source* has leaked. It enables to detect data leakage even if this data have been ciphered or encoded. An application that leaks data is not necessarily a malware, as data leakage is the business of both malware and user tracking frameworks in commercial applications — which constitutes essentially a large part of goodware applications. Whereas this feature gives useful insights on the application behaviors for manual analysis, its utility for automatic malware detection needs to be proved. Moreover the implementation and execution of taint tracking is costly, which leads us not to choose this feature for now.

Table 1: Comparative state of the art of dynamic analysis systems.

Reference	Tool name	Dynamic features used	App testing strategies	Objectives & comments
Thomas Bläsing et al. 2010	AASandbox	System calls (name)	Monkey	Data for malware/benign classification # Virtual device
Iker Burguera et al. 2011	Crowdroid	System calls (name)	Crowdsourced app interactions	Data for malware/benign classification # Real device
Cong Zheng et al. 2012	SmartDroid	Taint tracking, +?	UI brute force Restriction of execution to targeted activities	Data for classification or manual analysis # Virtual device
Lok Kwong Yan et al. 2012	DroidScope	System call (all content) Java calls (all content) Taint tracking	-	Data for classification or manual analysis # Virtual device
Vaibhav Rastogi et al. 2013	AppsPlayground	Taint tracking Targeted Android API Java calls	Monkey UI brute force Broadcast events Text fields filling	Malware/benign classification # Virtual device
Martina Lindorfer et al. 2014	Andrubis	App Java calls (all content) System calls (name, +?) Shared libraries targeted calls (name, +?) Taint tracking DNS/HTTP/FTP/SMTP/IRC (all content)	Monkey Broadcast events All possible app services All possible app activities	Data for classification or manual analysis # Virtual device
Mingyuan Xia et al. 2015	AppAudit	Taint tracking	∼	Malware/benign classification Data leaks detector # Symbolic execution
Vitor Monte Afonso et al. 2014	-	Targeted *Android* API Java calls (name) System calls (name)	Monkey Broadcast events	Malware/benign classification 96.66% accuracy # Virtual device
Kimberly Tam et al. 2015	CopperDroid	System calls (all content) Binder data	Broadcast events Text fields filling, +?	Data for classification or manual analysis # Virtual device
Gerardo Canfora et al. 2015	-	System call (name)	Monkey	Malware/benign classification 94.9% accuracy (unseen applications) # Virtual device
Marko Dimjašević et al. 2016	Maline	System call (name)	Monkey Broadcast events	Malware/benign classification 96% accuracy # Virtual device
Michelle Y. Wong et al. 2016	IntelliDroid	Taint tracking	Targeted inputs leading to suspicious Android API calls	Data for classification or manual analysis # Virtual device
Gerardo Canfora et al. 2016	-	Measures of resource consumption (CPU, Network, Memory, Storage I/O)	Monkey	Malware/benign classification 99.52% accuracy # Virtual device
-	Glassbox	Java calls (name) System calls (name) HTTP/HTTPS requests (all content)	Monkey UI brute force Broadcast events Real SMS/Call Text fields filling	Data for malware/benign classification # Real device

Table 2: Legend.

+?	The paper is not clear enough on those details and we cannot be sure that it is an exhaustive list
call (name)	Only the name of the call is used, in order to get the appearance frequency
#	Comment
-	No data
∼	Data exists but is irrelevant for this study

Java calls is another feature of interest as it captures an explicit behavior of the application. There are several ways to collect them:

- **Application Instrumentation** — This strategy does not need any modification of the *Android* source code and is not dependent on *Android* version. The application can be modified in order to dump targeted method parameters and return values. *APImonitor* (pjlantz, 2012) is a tool that enables the instrumentation of targeted Java calls. It reverses the application into *smali*, a human friendly format equivalent to the Java bytecode, with the *baksmali* (JesusFreke, 2009) utility. Then, it adds monitor routines around the targeted calls and it recompiles the code with the *smali* (JesusFreke, 2009) utility. This strategy is used by the authors of (Afonso et al., 2015).
- **DVM/ART Instrumentation** — The *DVM* (*Dalvik Virtual Machine*) or *ART* (*Android RunTime*, *Android* API version $\geq$ 4.4) is the system that interprets and executes all the application instructions. All Java calls converge to this component. Hence, by hooking the execution of *DVM/ART*, one can monitor and control all Java calls, their arguments and their return values. That implies the modification of *Android* source code and its compilation to a custom *ROM*. This is the strategy we chose to use for collecting Java calls. We prefer this method for keeping the application behaviors pristine, and particularly not inducing additional bugs. *Andrubis* (Lindorfer et al., 2014) and *DroidScope* (Yan and Yin, 2012) use similar approaches for tracing method calls.

For the last features, they are highly marginal. Here, *Andrubis* reported the retrieval of targeted shared library calls. Another data are the network communications. Only *Andrubis* reported the utilization of features from network communications, but without any further details. Our system makes use of *Panoptes* (Filiol and Irolla, 2015) for gathering plain text and encrypted web communications. A recent study, (Canfora et al., 2016), shows that the measures of resource consumption for malware detection is a promissing trail. It reports 99.52% accuracy with global measures of CPU usage, Network usage, RAM usage and Storage I/O usage.

2.2 Automated Testing Strategies

Dynamic analysis does not consist of launching the application and waiting the malware to show its malicious behaviors off. Malware are using logic bombs for hiding the payload. Logic bombs are a malicious piece of code that is executed after a condition is triggered. It means we need to test each application as a real user could have done it. For achieving this objective, several strategies have been used in the past:

- **Black Box Testing Strategies** — This class of strategies does not take the application source code into account, it focuses on sending inputs in the application without any prior information. This is the commonly used strategy. *Monkey* [1] is a dedicated tool created by *Google* for this task. It generates random events in a fast pace. Events range from system events (home/wifi/bluetooth/-sound volume etc.) to navigation events (motion, click). Because of its capacity to quickly explore applications activities, it has been used by most of dynamic analysis systems (*AASandbox* (Bläsing et al., 2010), *AppsPlayground* (Rastogi et al., 2013), *Andrubis* (Lindorfer et al., 2014), (Afonso et al., 2015), (Canfora and al., 2015 and 2016), *Maline* (Dimjašević et al., 2016)). *Monkey* is sometimes confused with *Monkey Runner* [2] in the literature, which is a python library for writing *Android* test routines.
- **White Box Testing Strategies** — This class of strategies takes the application source code into account. It focuses on sending specific inputs in the application for triggering targeted code paths. It requires the information from the static analysis of the application. Parsing the code is needed, to find the target methods and all their triggering conditions. *SmartDroid* (Zheng et al., 2012) and *IntelliDroid* (Wong and Lie, 2016) determine all paths to sensitive API calls, then execute one of the paths to the target with dynamic analysis. Another kind of *White Box* testing strategy is *symbolic execution* where dynamic analysis is done by simulating the execution of the application static code. *AppAudit* (Xia et al., 2015) uses this technique for finding data leaks with symbolic taint tracking.
- **Grey Box Testing Strategies** — This class of strategies partially takes the applications source code into account. It focuses on testing all visible inputs the application declares or displays (UI). It usually takes the output of the application to generate the next inputs. *Andrubis* uses a *Grey Box* strategy when it tests all possible application services and activities, because they get the information from the application manifest. *AppsPlayground* also uses *Grey Box* testing with its *Intelli-*

[1] https://developer.android.com/studio/test/monkey.html
[2] https://developer.android.com/studio/test/monkeyrunner/index.html

gent Execution where windows, widgets, and objects are uniquely identified to know when an object has already been explored. *Grey Box* testing have the advantages of using tests that the applications is suceptible to respond, contrary to *Black Box* testing, without the requirement of processing a Control Flow Graph, as *White Box* testing. It is then the most efficient way of testing applications. This is why we use strategy in Glassbox.

2.3 Real Devices

The use of real devices for dynamic analysis started with *Crowdroid* (Burguera et al., 2011), a crowdsourced based analysis. Whereas this approach give good results, one cannot ask users to execute real malware on their personal device. So this system can only be an option, when we have already a trained machine learning algorithm, to find malware in the wild.

BareDroid is a system which manages real devices in large scale for dynamical analysis. Whereas *BareDroid* (Mutti et al., 2015) cannot be considered as a dynamical analysis system, because it does not analyse applications, it brought two major results for our study. First, real devices for dynamic analysis systems are a scalable solution financially and in excution time compared to virtual devices. Second, using real devices drastically improves features detected for malware families that often rely on emulator evasion like *Android.HeHe*, *Android Pincer*, and *OBAD*.

3 ARCHITECTURE OVERVIEW

Glassbox (Figure 1) is a modular system distributed among one or several phones and a computer. Each part is detailed in the following sections.

3.1 Android Instrumentation

A custom *Android* OS has been made, based on the *Android Open Source Project (AOSP)* [3]. The objective here is to log dynamically each Java call of a targeted application. This involves hooking these calls, at a point where all of them pass through. We instrumented *ART (Android RunTime)* [4], the *Android* managed runtime system that executes application instructions. With the default parameters, we found that *ART* (at least until *Android Marshmallow*) have the following important behaviors for our study:

- The first time *Android* is launched, Java *Android* API libraries and applications are optimized and compiled to a native code format called *OAT* (Sabanal, 2015).
- Each time a new application is installed, it is optimized and compiled to *OAT* format.
- Java methods can be executed in three ways: by an *OAT JUMP* instruction to the method address, by the *ART* interpreter for non-compiled methods (debugging purposes mostly), or via the *Binder* for invoking a method from another process or with Java *Reflection*. Details on the *Android Binder* can be found in (Schreiber, 2011) chapter 4.

A straightforward way of hooking Java calls is to instrument the *ART* interpreter. Unfortunately only a few calls are executed through it because most of the code is compiled into *OAT* and therefore it is not interpreted. We forced all calls to be interpreted by disabling several optimizations. The first one is the disabling of the compilation to *OAT*. That leads calls to be interpreted before executed. But others optimizations mechanisms comes into play, namely *direct branching* and *inlining*.

The boot classpath contains the *Android* framework (Figure 2) and core libraries. They are always compiled in *OAT* resulting in a *boot.oat* file. This file is mapped into memory by the *Zygote* process, started at the initialisation of *Android*. For launching an application, the main activity is given at *Zygote* in parameters. When *Zygote* is called that way, it forks and starts the given activity. It means that any application has access to same instance of the *Android* framework and core libraries. *Direct branching* is an optimisation that replaces framework/core method calls by their actual address in memory. So the calls does not pass through the interpreter. That optimisation is disabled.

Then *inlining* is an optimisation that replaces short and frequently used methods with their actual code. Although, it slightly increases the application size in memory, runtime performance are increased. As there is no method any more, it cannot be hooked in the *ART* interpreter. That optimisation is also disabled.

A monitoring routine is added to *ART* interpreter that logs any method call from a targeted application *pid*. A sample of a capture of Java calls is shown in annexes. All these modifications overload the global execution of *Android*. Whereas it is not noticeable for most of the applications, on gaming applications are visibly slow down by this approach.

Lastly the phone is shipped with a real *SIM* card for luring malware payloads with SMS/MMS/Calls.

[3] https://source.android.com/

[4] https://source.android.com/devices/tech/dalvik/index.html

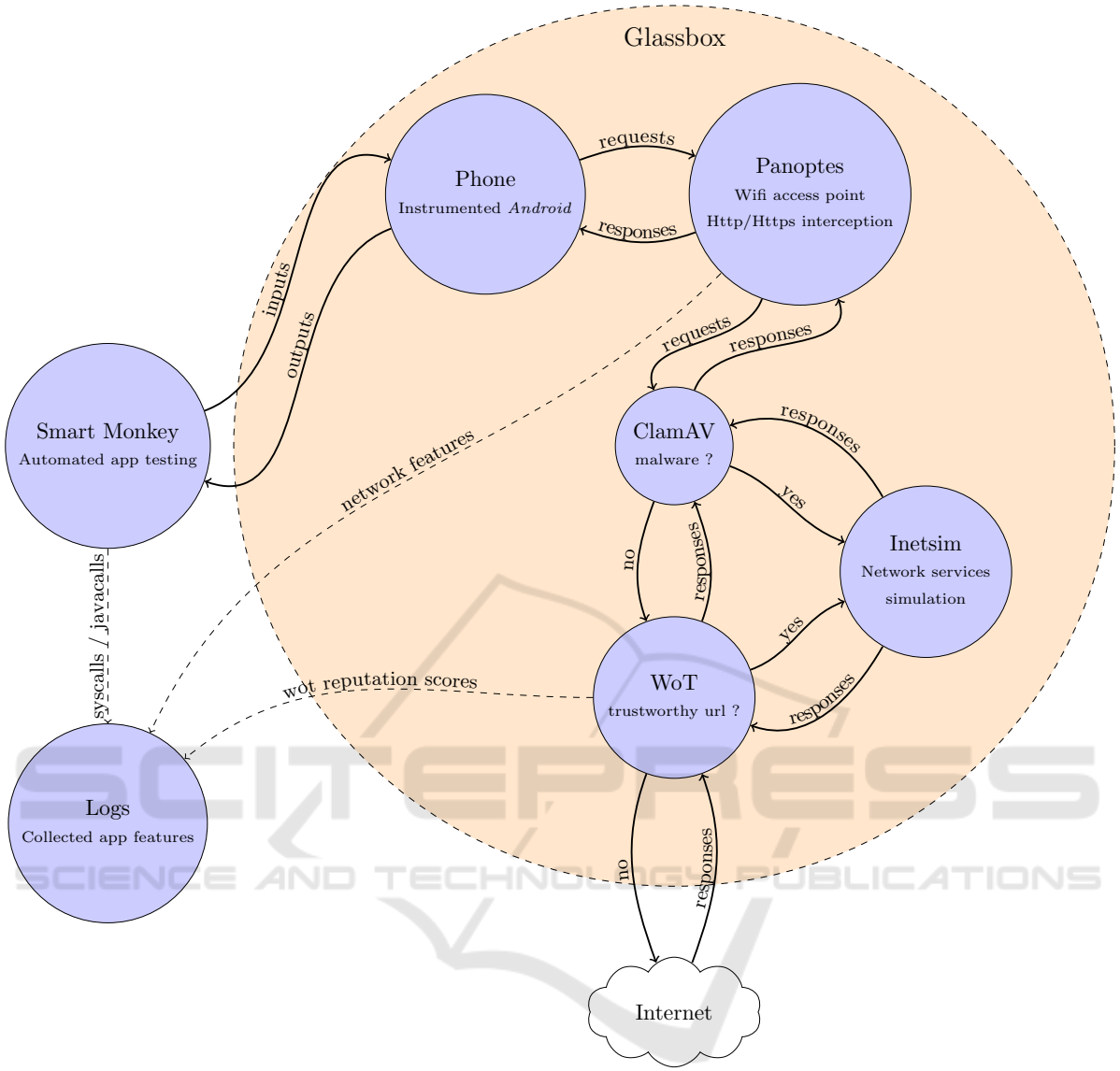

Figure 1: Glassbox — Architecture overview.

Many malware may use it for stealing money with premium numbers, and because we use a real *SIM* card it would actually cost us money. We modified the telephony framework of the *Android* API to reject all outgoing communications except for our own phone number. When a forbidden call is made, the calling UI pops and closes after one second around. This way does not crash applications that rely on calls and SMS.

3.2 Network Control & Monitoring

All communications of the instrumented phone pass through a transparent *SSL/TLS* interception proxy behind a wifi access point. This is set by *Panoptes* (Filiol and Irolla, 2015). To understand how it works we need to describe a part of the *TLS* handshake. Here is the regular behavior of a *https* request on *Android*:

Android have a keystore of all root certificates the system trusts. When a *SSL/TLS* request is initialized, the requested server send its certificate. It contains identifying informations — like the domain name that must verify the contacted domain name — and a signature that can only be decrypted with the right root CA. The server certificate is tested with each trusted root CA, and if one matches the communication is accepted. Extended information on the *TLS* handshake can be found with the RFC 2246 memo (Dierks and Allen, 1999).

For our interception system to work, a *SSL/TLS*

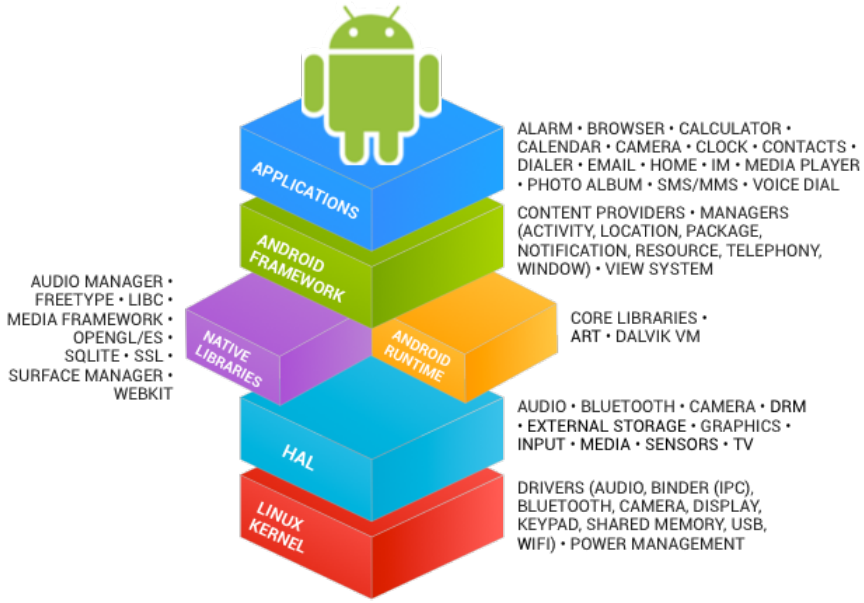

Figure 2: Android architecture overview.

root certificate from a custom certification authority (CA) is implanted in the keystore of *Android*. When the device requests a *https* web page, the request goes through the proxy. It is parsed and a new one is initialised to be sent to the original recipient. The response is encapsulated in a new *SSL/TLS* response signed by our custom certificate. This custom certificate is dynamically generated with the recipient identifying informations and our custom root CA private key. As the communication is signed by an authority of certification that is known by the client, *Android* accepts it without any warning. Finally, all *HTTP/HTTPS* communications are logged and a report can be generated which is convenient for manual analysis if needed.

This system has been extended to support manipulation of requests. The objective is to restrict the proliferation of malware and the damage that it may produce. As *Glassbox* runs malware, it may have a negative impact on its environment. An extreme mean could be to disconnect the system from internet but we would see less or no malicious behavior at all for numerous applications. Our design is a trade-off between safety and behavior detection:

- *ClamAV* (Kojm, 2004) is used to detect known malware sent through network. If a malware is detected, the payload is removed from the request and is redirected to *Inetsim* (Hungenberg and Eckert, 2013), a network services simulation server that replies consistently to the requests. It forbids the communication between the application and internet without crashing it.
- For all other requests, we assess the reputation of the domain name or *IP* address with the *Web of Trust (WoT)* [5] API. *WoT* is a browser extension that filters urls based on different reputation rating. These rating come mainly from the users. If the request contacts a known address with a good reputation, we forbid the application under test to reach it, then it is redirected to *Inetsim*. The advantages are twofold. The application cannot damage a respectable website, and it pre-filters behaviors for classification.

Finally features from communications content are collected and the *WoT* reputation scores as well. A sample of a network capture is shown in the annexes.

3.3 Automated Application Testing

Smart Monkey is an automated testing program based on *Grey Box* strategies. The context of the application is determined at runtime for the automatic exploration. We use *UIautomator* [6], a tool that can dump the hierarchy tree of the current UI elements present on screen. It enable us to monitor variables of each UI element at runtime. For a smart exploration, we need to know if we have already processed an element. Unfortunately, elements do not carry such a unique identifier. Nonetheless, we found that elements can be identified to some degree:

[5]https://www.mywot.com/wiki/API

[6]https://stuff.mit.edu/afs/sipb/project/android/docs/tools/help/uiautomator/index.html

- **Strong Identification** — Elements can have an associated *ID string* that developers set. Concatenated to the current activity name we have robust identification, but for most of the elements this field is empty. With the same method, a *content description* is sometimes associated with the element. We can also use this for strong identification.
- **Partial Identification** — If we do not have access to the previous values, which happens most of the times, we can use lesser discriminative values. Textfields can be set with an initial value, or a printed text can be associated to it. With no better available options we use the element dimensions to identify it. Obviously, when an element is partially identified, a risk of false positive is possible.

Moreover each element carries a list of actions it can trigger. Our automatic exploration consists of systematically triggering all actions of all elements for all activities. We do not try each combination of actions as it would not scale and be mostly redundant. To this basic general process we add targeted actions to trigger more sophisticated behaviors:

- Some textfields of interest are detected like phone number, first or last name, email address, IBAN, country, city, street addresses, password or pin code. These textfields are filled with consistent values accordingly. For this task, we use databases of realistic data (samples can be found in the annexes). Uncategorised textfields are filled with a pseudo-random string.
- The order of actions done matters. For example login and password textfields must be filled before validating. In the exploration, filling textfields and check-boxes takes precedence over the rest.
- An application can register a receiver for a broadcast *Android* event like the change of phone state or wifi state. It can be done statically in the application manifest or dynamically. Those dynamical receivers could be hidden from static analysis with obfuscation. To trigger the receivers code, we test applications with a list of broadcast events that are often used by malware (a partial list is given in annexes). Moreover, real SMS and phone calls are sent to the real device own number.

We finally use the *Monkey* program during the analysis. It can help to trigger behaviors requiring complex inputs combination that *Smart Monkey* could miss. At the end comes the cleaning phase. For our real device we keep a white list of regular processes and installed applications — regular, system and device administrator applications. Non-authorised processes are killed and applications uninstalled. Important phone configurations like wifi, data network and sounds are reset to a predefined value.

4 EXPERIMENTATION

4.1 Performance Measure

We use the *average coverage of basic blocks* for quantifying the performance of the application code coverage. It is a measure of the performance of the *Smart Monkey* component of *Glassbox*. Here are definitions of the vocabulary used in the experimentation:

- A *basic block* is an uninterrupted section of instructions. A *basic block* begins at the start of the program or at the target of a control transfert instruction (JUMP/CALL/RETURN). It ends at the next control transfert instruction.
- The *basic block coverage* for an application is the number of unique basic blocks executed at runtime divided by the number of unique basic blocks present in the source code.
- The *average coverage of basic blocks* is the sum of the *basic block coverage* of all applications divided by the number of applications.

4.2 Dataset

We use the *AndroCoverage Dataset* (AndroCoverage, 2016) for our experimentation. It contains 100 applications from *F-Droid* [7], which is a repository of free and open source (*FOSS*) applications. We have manually selected them with the following criteria for each application:

- It does not depend on a third party library or application as an automatic tool would be unable to install it.
- It does not depend on root privilege. To meet the requirement of a maximum of testing tools configuration, we stick with regular privilege.
- It does not depend on local or temporary remote data. We want the application to be usable worldwide and in the long-term. This category excludes applications for a temporary event or a specific country.

Our goal is to use applications which show a large variety of different and steady behaviors. It is why we predict that performance on the *AndroCoverage*

[7] https://f-droid.org/

Dataset will be overestimated compared to the average of real applications. This dataset is to be used to compare the performance of different automated testing tools on the same ground.

The *AndroCoverage Dataset* is supplied with tools which instruments the application, adding monitoring routines for code coverage. These tools are partially founded on *BBoxTester* (Zhauniarovich et al., 2015), a tool for measuring the code coverage for *Black Box* testing of *Android* applications.

4.3 Methodology

Research community used different strategies for automated application testing, with different evaluation methods and different datasets. To promote the successful strategies for future researches on the domain, we need a standard for the experimentation. Otherwise, we cannot compare the results objectively. The research titled *Automated Test Input Generation for Android: Are We There Yet?* (Choudhary et al., 2015) shows the re-evaluation on the same ground of 5 published automated testing tools for *Android*. The experimental results found is far from what have been claimed in the published papers. Moreover, according to this study, the *Monkey* program have the best performances above all at around 53% *average coverage of statements* on 68 selected applications. Either all rechearches on *Android* automated testing are not better than a random event generator or the evaluation methodology lacks pertinence. Our opinion is that a better methodology can highlight the contribution of main researches to the field.

To summarize, these observations reveal several problems on the experimental results:

- (1) They are currently not reproductible.
- (2) They cannot be compared to each other.
- (3) They do not hightlight the contribution of the evaluated testing method compared to *Monkey* program.

To answer those problems, we propose the following rules:

- (2)[8] A **common performance measure**. We propose the *average coverage of basic blocks*. *Statement coverage* (also called *line coverage*) is considered as the weakest code coverage measure by specialists in software testing. This metric should not be used when another one is available. For an argued reflection about coverage metrics, we refer to the paper *What is Wrong with Statement Coverage* (Cornett, 1999).
- (1)(2) A **common dataset** and **common tools** for instrumenting the applications. We propose the *AndroCoverage Dataset* (AndroCoverage, 2016).
- (1)(2) A **common configuration** — *Monkey* arguments, a fixed seed for every random number generator used and application versions. These information are either present in annexes of this document or on the *AndroCoverage* Github web page.
- (3) To assess the performance of the combination of both *Monkey* and the evaluated testing method (in our case it is *Smart Monkey*). Compared separately, the performance of *Monkey* and the evaluated method does not hightlight new code paths that have been triggered by the evaluated method. A complex method would not seem successful whereas it would have triggered complex conditions that *Monkey* could never find. Moreover, the *Monkey* program is embeded in every *Android* device (real and virtual), it interacts in a very fast pace with the application and produces good results. Then, on an operational situation, it makes sense to use it in addition to any research tool.

4.4 Results

Smart Monkey usually runs *Monkey* at the beginning of the analysis. For a fair trial, we tested the performance of its code coverage with and without *Monkey*. The configuration of the *Monkey* tool has been described in annexes. The results are presented on Table 3.

The *Monkey* program tends to generate bugs with the instrumentation process. For a significant amount of applications (16%) we are unable to get the coverage rate. We note that the same applications crash between *Monkey* and *Smart Monkey* so the crash rate has no effect on the performance comparison between both programs.

The raw results does not give enough insight of the contribution of *Smart Monkey*. We are interested in the new paths that have been triggered compared to the *Monkey* program. Therefore we calculate the increase of *average coverage of basic blocks* of *Smart Monkey* compared to *Monkey*:

$$\frac{smartmonkey_{ac}}{monkey_{ac}} = 1.1352$$

Where:

- $smartmonkey_{ac}$ is the average coverage of basic blocks of Smart Monkey.
- $monkey_{ac}$ is the average coverage of basic blocks of Monkey.

[8]This number and the following ones refers the problem number it solves

Table 3: Code coverage results.

Method	Classes average coverage	Methods average coverage	Blocks average coverage	Crash rate
Monkey	32.93%	35.05%	36.32%	16%
Smart Monkey (w/o Monkey)	34.84%	36.68%	37.73%	0%
Smart Monkey (with Monkey)	37.12%	41.6%	41.23%	16%

It means that the testing strategy we have set up in *Smart Monkey* leads to average increase of 13.52% of *basic blocks coverage*.

5 LIMITATIONS

Dynamic analysis systems that allows internet communications are vulnerable to fingerprinting. Our platform is not an exception. For example *Bouncer* (Lockheimer, 2012) have been the target of remote shell attacks (Percoco and Nicholas, 2012) that enabled the fingerprinting of the system. The malware gets some information on the system and sends it to a command and control server. Hence, the malware author can reshape the trigger conditions of the logic bomb. We accepted this risk for now. A solution halfway between shutting down all communications and no filtering at all could be to strip all outgoing information — POST request contents/GET url variables/Cookies/Metadata fields. This could lead to a loss of behaviors and the negative impact of such solution needs to be measured. Anyway a smart malware author will eventually find a way to leak remote shell outputs.

The network monitoring has limitations. First, it cannot currently handle all protocols like *POP*, *IMAP* and *FTP* so these protocols are simply blocked. In fact the communications are parsed, to get its content, the destination and the metadata. So this parsing needs to be changed for each protocol. It is an impossible task of adding one by one all protocols, so we would need to measure the protocol usage and implement the most used ones. At last, there is a countermeasure to our *SSL/TLS* interception, namely *certificate pinning*. The requipement of the interception is the implantation of a custom root certificate in the *Android* keystore of trusted certificates. An application can choose to discard the *Android* keystore and to embed its own. Therefore when a communication, encrypted with our custom certificate, is checked, the communication is rejected. This technique is used in many banking applications (Filiol and Irolla, 2015). In fact the point of view of the bank is: the user *OS* cannot be trusted. Although we have no evidence that it happens for malware, it may be used by an avant-gardist malware and other would follow the trail. It is inconvenient for malware authors to buy a certificate signed by an authority of certification, as a payment trace could identify them. Despite of that, it is possible to get a valid certificate from *Let's Encrypt* [9], or to control a legtimate server via hacking and use it as a relay for the *C&C* server. In these cases, *certificate pinning* could be used for hiding communications from analysts or interception systems. A counter to this technique is instrumentation. By monitoring arguments of the *SSL/TLS* encryption method, one can get the plaintext communications. We have done it manually for some banking applications (Filiol and Irolla, 2015) with *APImonitor* (pjlantz, 2012), but doing it automatically is another issue. Applications that use *certificate pinning* generally embed their own library for *SSL/TLS* encryption, so detecting dynamically which call is the *SSL/TLS* encryption method can be challenging.

Last, the cleaning phase of *Glassbox* fits the security needed for a prototype. However, to move to an operational situation with malware that could execute 0-day root exploits, we need a real factory-reset of the phone. This is why we plan to integrate the open source project *BareDroid* as a part of *Smart Monkey*, for its factory-reset capability on real device.

6 CONCLUSION & FUTURE RESEARCH

This paper contributes to the domain of dynamic analysis system for *Android* in three ways. First, we presented *Glassbox* a functional prototype of a platform that uses real devices, controls network and GSM communications to some extends and monitors Java calls, systems calls and network communication content. Second, we experimented *Smart Monkey*, an automatic testing tool with a *Grey Box* testing strategy. We showed that it enhances the application code coverage compared to the common *Black Box* testing tool called *Monkey*. Last, we presented a method of evaluation of automated testing tools to research community. This method covers the problems of reproducibility, the comparison with other works and of the contribution measurement of the tool. We made

[9]https://letsencrypt.org/

the dataset available on Github under the name *AndroCoverage*.
The next step is to use *Glassbox* on malware/benign applications and to use the features found on a machine learning algorithm. We are working on the classification of these data with a neural network.

REFERENCES

Afonso, V. M., de Amorim, M. F., Grégio, A. R. A., Junquera, G. B., and de Geus, P. L. (2015). Identifying android malware using dynamically obtained features. *Journal of Computer Virology and Hacking Techniques*, 11(1):9–17.

AndroCoverage (2016). Androcoverage dataset. [Online] https://github.com/androcoverage/androcoverage.

Bläsing, T., Batyuk, L., Schmidt, A. D., Camtepe, S. A., and Albayrak, S. (2010). An android application sandbox system for suspicious software detection. In *Malicious and Unwanted Software (MALWARE), 5th International Conference on*, pages 55–62.

Burguera, I., Zurutuza, U., and Nadjm-Tehrani, S. (2011). Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 15–26. ACM.

Canfora, G., Medvet, E., Mercaldo, F., and Visaggio, C. A. (2015). Detecting android malware using sequences of system calls. In *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, pages 13–20. ACM.

Canfora, G., Medvet, E., Mercaldo, F., and Visaggio, C. A. (2016). Acquiring and analyzing app metrics for effective mobile malware detection. In *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics*, pages 50–57. ACM.

Choudhary, S. R., Gorla, A., and Orso, A. (2015). Automated test input generation for android: Are we there yet?(e). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 429–440. IEEE.

Cornett, S. (1999). What is wrong with statement coverage. [Online] http://www.bullseye.com/statementCoverage.html.

Dharmdasani, H. (2014). Android.hehe: Malware now disconnects phone calls. [Online] https://www.fireeye.com/blog/threat-research/2014/01/android-hehe-malware-now-disconnects-phone-calls.html.

Dierks, T. and Allen, C. (1999). The tls protocol version 1.0. [Online] http://www.ietf.org/rfc/rfc2246.txt.

Dimjašević, M., Atzeni, S., Ugrina, I., and Rakamaric, Z. (2016). Evaluation of android malware detection based on system calls. In *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics*, pages 1–8. ACM.

Filiol, E. and Irolla, P. (Black Hat Asia 2015). (in)security of mobile banking... and of other mobile apps.

Hungenberg, T. and Eckert, M. (2013). Inetsim: Internet services simulation suite.

IDC (2015). Smartphone os market share, 2015 q2. [Online] http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

JesusFreke (2009). Github - smali readme. [Online] https://github.com/JesusFreke/smali.

Jing, Y., Zhao, Z., Ahn, G.-J., and Hu, H. (2014). Morpheus: automatically generating heuristics to detect android emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 216–225. ACM.

Kojm, T. (2004). Clamav.

Lindorfer, M., Neugschwandtner, M., Weichselbaum, L., Fratantonio, Y., v. d. Veen, V., and Platzer, C. (2014). Andrubis – 1,000,000 apps later: A view on current android malware behaviors. In *2014 Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, pages 3–17.

Lockheimer, H. (2012). Android and security. [Online] http://googlemobile.blogspot.fr/2012/02/android-and-security.html.

Mutti, S., Fratantonio, Y., Bianchi, A., Invernizzi, L., Corbetta, J., Kirat, D., Kruegel, C., and Vigna, G. (2015). Baredroid: Large-scale analysis of android apps on real devices. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 71–80. ACM.

Percoco and Nicholas, J. (2012). Adventures in bouncerland.

pjlantz (2012). Droidbox - apimonitor.wiki. [Online] https://code.google.com/archive/p/droidbox/wikis/APIMonitor.wiki.

Rastogi, V., Chen, Y., and Enck, W. (2013). Appsplayground: automatic security analysis of smartphone applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 209–220. ACM.

Sabanal, P. (2015). Hiding behind art.

Schreiber, T. (2011). Android binder - android interprocess communication.

Tam, K., Khan, S. J., Fattori, A., and Cavallaro, L. (2015). Copperdroid: Automatic reconstruction of android malware behaviors. In *NDSS*.

Wong, M. Y. and Lie, D. (2016). Intellidroid: A targeted input generator for the dynamic analysis of android malware.

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

Yan, L. K. and Yin, H. (2012). Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 569–584.

Zhauniarovich, Y., Philippov, A., Gadyatskaya, O., Crispo, B., and Massacci, F. (2015). Towards black box testing of android apps. In *2015 Tenth International*

Conference on Availability, Reliability and Security (ARES), pages 501–510.

Zheng, C., Zhu, S., Dai, S., Gu, G., Gong, X., Han, X., and Zou, W. (2012). Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 93–104. ACM.

ANNEXES

Data Samples Used in Smart Monkey

```
$> head random-iban.txt
AL94283405797977629281563659
AL60726122350056756457999447
AL23793884960503665784521815
AL91081264763546250859672884

$> head broadcast-events.txt
android.intent.action.BOOT_COMPLETED
android.intent.action.BATTERY_CHANGED
android.net.conn.CONNECTIVITY_CHANGE
android.intent.action.USER_PRESENT
```

Sample of a Java Calls Capture

```
$> logcat
[...]
void java.lang.StringBuilder.<init>
java.lang.StringBuilder java.lang.
    StringBuilder.append
java.lang.StringBuilder java.lang.
    StringBuilder.append
java.lang.String java.lang.
    StringBuilder.toString
void com.energysource.szj.android.Log.i
android.os.Looper android.os.Looper.
    getMainLooper
[...]
```

Sample of a Network Capture

```
[...]
<header>
  <method>R0VU</method>
  <scheme>aHR0cA==</scheme>
  <host>MTE1LjE4Mi4zMC42OA==</host>
  <port>ODA=</port>
  <path>L0dldEluZm8uYXNoeD9hcHBpZD03Z
    mZjN2JlOTJmM2M0YTdmYTA4MzUxZTNkNT
    NmOThkYSZhcHB2ZXI9Mjc2JnY9MS4wLjQ
    mY2xpZW50PTImcG49Y29tLmdwLnNlYXJj
    aCZ1c2VydmVyPTIuMCZhZHR5cGU9MiZjb
    3VudHJ5PWZyJm50PTImbW5vPTIwODE1Jn
    V1aWQ9ZmZmZmZmZmYtZWIwOS05NDcwLTU
    zY2UtYmMxYjAwMDAwMDAwJm9zPTYuMC4x
    JmRuPUFPU1Arb24rSGFtbWVySGVhZCZza
    XplPTEwODAqMTc3NiZjYz00JmNtPTM4Lj
    QwJnJhbT0xODk5NTA4a2I=
  </path>
  <http_version>
    SFRUUC8xLjE=
  </http_version>
  <host>Y2ZnLmFkc21vZ28uY29t</host>
  <Connection>
    S2VlcC1BbGl2ZQ==
  </Connection>
  <User-Agent>
    QXBhY2hlLUh0dHBDbGllbnQvVU5BVkFJT
    EFCTEUgKGphdmEgMS40KQ==
  </User-Agent>
</header>
<content />
[...]
```

NB: all field values are encoded in base 64

Monkey Configuration

```
monkey -s 0 --pct-syskeys 0 --pct-
    appswitch 0 --throttle 50 -p <
    package-name> -v 500

-s 0: The seed of the random number
    generator is fixed to 0
--pct-syskeys 0: No system key events
    are sent, such as Home, Back, Start
     Call, End Call, or Volume inputs.
--pct-appswitch 0: No startActivity()
    are issued as calling the
    instrumentation activity another
    time breaks it.
--throttle 50: The delay between events
     is fixed to 50 milliseconds.
500: A total of 500 events are sent.
```