POLITECNICO DI MILANO
Scuola di Ingegneria Industriale e dell' Informazione
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

DriveDroid: a Remote Execution Environment and UI Exerciser for Android Malware Analysis

Relatore: Prof. Federico Maggi
Correlatore: Prof. Stefano Zanero

Tesi di laurea di:
Claudio Rizzo Matricola n. 800471
Emanuele Uliana Matricola n. 799256

Anno Accademico 2014-2015

"Nell'attività fisica, così come in quella intellettuale, l'eccellenza si raggiunge al 10 per 100 con l'ispirazione, ma al 90 per 100 con la sudorazione. Non esistono i campioni e i geni "naturali": esistono gli individui dotati, che si fanno "artificialmente" un mazzo tanto per mettere a frutto le loro doti".

Piergiorgio Odifreddi, a proposito di Pietro Mennea

Sommario

Android è senza dubbio il sistema operativo più diffuso nel mercato dei dispositivi mobili (smartphone e tablet): grazie alla sua natura open source e alla sua conseguente adattabilità a diversi tipi di hardware, detiene (dato aggiornato al secondo quadrimestre del 2015) l'82.5% delle quote di mercato. Gli utenti possono scaricare applicazioni destinate a sistemi Android sia dallo store ufficiale di Google, il Play Store, sia da svariati market di terze parti. Il successo della piattaforma ha fin da subito attirato l'attenzione dei malware writers i quali utilizzano svariate tecniche per rendere disponibili le loro applicazioni malevole. Una delle più famose consiste nell'utilizzare strumenti open source pubblicamente disponibili per decompilare applicazioni legittime, inserire codice malevolo, ricompilarle e pubblicarle su uno o più market. Questo metodo si è rivelato più volte efficace per aggirare i controlli di sicurezza che i market, Play Store compreso, effettuano sulle applicazioni prima di avallarne la pubblicazione.

Per questo motivo l'analisi di mobile malware è diventata rapidamente un campo di ricerca interessante e, sulla falsa riga dell'analisi di desktop malware, due sono gli approcci principali: analisi statica e analisi dinamica. L'analisi statica permette di esaminare un'applicazione senza eseguirla e, notoriamente, è apprezzata per gli alti livelli di code coverage raggiunti. Tuttavia, in caso di malware "protetti", (tramite compressione, offuscamento, ecc.) l'analisi statica è inefficace e applicazioni malevole di questo tipo sono sempre più frequenti. L'analisi dinamica, d'altro canto, avviene durante l'esecuzione vera e propria dell'applicazione, forzando quindi la decompressione/decrittazione e il deoffuscamento delle parti di codice originariamente protette, superando quindi i principali limiti dell'analisi statica, però ha i noti svantaggi di poter mostrare solamente i comportamenti (malevoli o non) derivati dai path del codice effettivamente esplorati. Inoltre necessita sia di un apposito approccio di stimolazione automatica, sia di un ambiente emulato e instrumentato che, a volte, presenta delle discontinuità con l'ambiente hardware su cui le applicazioni sono normalmente eseguite. I malware writers hanno imparato sia a far riconoscere ai propri malware gli ambienti emulati (con tecniche fortunatamente prevedibili e a cui si può facilmente porre rimedio), sia a presentare all'utente, umano o automa che sia, delle "sfide interattive" (come il classico giochino "clicca sul tal oggetto per vincere un premio") che i sistemi di stimolazione automatici non sono in grado di superare. Ciò diventa problematico quando i comportamenti malevoli vengono espletati solamente dopo il superamento di queste sfide.

Una delle sfide dell'analisi dinamica è progettare un sistema di stimolazone di applicazioni che possa efficacemente sostituire un utente umano. Un primo tentativo è stato effettuato con i cosiddetti stress-tools casuali, ovvero degli strumenti che iniettano nell'applicazione da testare una serie di eventi di input (touch ad esempio) generati in modo casuale in

posizioni casuali e con pause molto brevi tra ciascuno di essi, in modo da raggiungere un alto numero di eventi in un tempo ragionevole. Il sostanziale problema con questo approccio è la scarsa code coverage dovuta al fatto che, non essendo in grado di imitare i comportamenti umani di fronte ad un'interfaccia grafica, non riescono a cogliere il significato semantico degi elementi visualizzati. Inoltre gli stress-tools casuali non risolvono il problema delle sfide interattive. Di recente è stato progettato un sistema, PuppetDroid che cerca di porre rimedio al primo problema: l'idea di fondo è che un umano stimola un'applicazione più a fondo di un tool automatico, quindi l'approccio di questo lavoro è di sfruttare l'utente umano per stimolare un certo numero di applicazioni, raccogliere gli eventi generati da tale stimolazione e reiniettarli su appliczioni dalla grafica simile a quella dell'applicazione di partenza. Tuttavia PuppetDroid non si preoccupa di costruire un modello dell'applicazione target, né dei dati raccolti dalla stimolazione e ciò può portare all'arrivo in punti in cui è impossibile procedere con la stimolazione o, peggio, in cui l'applicazione smette di funzionare a causa di una serie di input ingestibili.

Partendo dall'inutizione alla base di PuppetDroid, ne abbiamo sviluppato un'estensione, chiamata DriveDroid, che costruisce un modello di una applicazione e degli eventi iniettati su di essa da un utente che con essa interagisce. Questo modello è un grafo orientato in cui i nodi rappresentano gli stati dell'interfaccia grafica dell'applicazione, mentre gli archi rappresentano le interazioni che consentono le transizioni di stato. Inoltre abbiamo progettato una nuova strategia per "rieseguire" i grafi registrati su applicazioni graficamente simili, ottenute tramite iniziale clustering di un dataset e successivo lookup nei cluster ottenuti, moltiplicando in media di un fattore 5 (ma anche di qualche decina nel migliore dei casi) il numero di applicazioni che possono esere testate al costo della registrazione da parte di un utente di un singolo grafo. Abbiamo validato il nostro approccio su un dataset di 1028 applicazioni Android, dimostrando che in media funziona almeno tanto bene quanto lo stato dell'arte e, in più, permette di superare casi di evasione di analisi dinamica effettuati tramite sfde interattive. I risultati mostrano che l'idea di trasformare DriveDroid in un sistema pubblico che permetta di raccogliere grafi su larga scala, per poi rieseguirli con il nostro approccio, è promettente.

Abstract

Android malware authors keep improving their malicious applications, giving them the capability to evade automatic analysis. Evasion mechanism can range from simple emulation-detection techniques, which are fortunately easy to counter-evade, to advanced interaction tests that determine whether a human user or an automated analysis is interacting with the malware, and refuse to execute in the latter case.

In contrast, researchers proposed many approaches to enhance malware analysis. An emergent, promising approach consists in exercising the user interface (UI) of an Android application in order to trigger, and thus be able to analyse, as many (malicious) behaviours as possible. The idea is that UI interactions are recorded and automatically re-executed on applications with a similar UI. In our work, we bring this approach one step further, changing the way UI interactions are recorded and re-executed, and engineering the whole system from the ground up. Our main contribution consists in how we model an Android application as a graph and how graphs obtained from an app are used to drive the re-execution of another app with a similar UI. More precisely, we allow graphs obtained by several apps to be combined together to form a "super graph," and we designed a UI-based lookup to find similar apps in a fast way.

We developed our approach and we conducted experiments over a dataset of 1,028 applications. The results show that our new approach can reach at least the same code coverage than the state-of-the-art tools and that it allows to overcome some limitations of the state of the art.

Contents

Sommario ii

Abstract iv

Contents v

List of Figures vii

1 Introduction 1

2 Background and motivation 3
- 2.1 Android . . . 3
  - 2.1.1 Android user interface . . . 4
  - 2.1.2 Android events management . . . 5
  - 2.1.3 Android malware . . . 6
- 2.2 Stimulation of Android applications . . . 6
  - 2.2.1 Exercising Android applications . . . 7
  - 2.2.2 Motivating example . . . 9
  - 2.2.3 Limitations . . . 10
- 2.3 Goals and challenges . . . 11

3 DriveDroid 13
- 3.1 Approach overview . . . 13
  - 3.1.1 Phase 1: Clustering . . . 13
  - 3.1.2 Phase 2: Recording . . . 14
  - 3.1.3 Phase 3: UI-similarity graph lookup . . . 15
  - 3.1.4 Phase 4: Re-execution . . . 16
- 3.2 Implementation details . . . 17
  - 3.2.1 Similarity and lookup . . . 17
  - 3.2.2 DriveDroid graph . . . 19
- 3.3 System architecture overview . . . 22
  - 3.3.1 PuppetApplication . . . 22
  - 3.3.2 Worker . . . 24
- 3.4 Technical details . . . 24
  - 3.4.1 Main Server . . . 24
  - 3.4.2 Worker . . . 25
    - 3.4.2.1 AVD . . . 25

3.4.2.2 VNC Server . . . 25
3.4.3 Recorder . . . 26
3.4.4 Rerunner . . . 26
3.4.5 Web application . . . 26
3.4.6 Public key infrastructure . . . 26
3.4.7 Persistent data management . . . 27
3.4.8 Use cases . . . 27

4 Experimental evaluation **34**
4.1 Dataset . . . 34
4.2 Experiment 1: Clustering and lookup evaluation (Phase 1 and 3) . . . 35
4.2.1 Experimental set-up . . . 35
4.2.2 Clustering evaluation . . . 35
4.2.3 Lookup performance evaluation . . . 36
4.3 Experiment 2: Recording and re-execution evaluation (Phase 2 and 4) . . . 37
4.3.1 Experimental set-up . . . 37
4.3.2 Human vs. Monkey . . . 39
4.3.3 Re-execution vs. Monkey . . . 40
4.4 Experiment 3: Interactive challenge evaluation . . . 41
4.4.1 Dataset . . . 41
4.4.2 Experimental set-up . . . 41
4.4.3 Results . . . 41

5 Limitations and future work **42**
5.1 Approach limitations . . . 42
5.2 Implementation limitations . . . 43
5.3 Future work . . . 44

6 Conclusions **45**

Bibliography **46**

List of Figures

2.1 Android view hierarchy: abstract representation 4
2.2 Android view hierarchy: detailed representation 5
2.3 Interactive challenge: right and wrong stimulation of our application . . . 10

3.1 Approach: high level overview of the 4 phases 14
3.2 Phase 2 (Graph recording): visual representation of a graph with app screenshots 15
3.3 Phase 3 (UI-similarity graph lookup) 16
3.4 Stuck node: example of stuck node leading to another stuck node 17
3.5 Recording: example of transitions in the recording phase 20
3.6 System Architecture: system high level view 23
3.7 Public key infrastructure: root CA, client, servers and workers 27
3.8 Database: database collections schemas 28
3.9 Registration: client registration flow 29
3.10 APK search: APKs search flow 30
3.11 Test session: test session flow 32
3.12 Re-execution: re-execution flow 33

4.1 Tuning clusters number: knee-elbow to decide the number of clusters . . . 35
4.2 Clustering evaluation: performance and average size 36
4.3 Lookup evaluation: naïve and centroid approaches performance evaluation 38
4.4 Monkey vs Human: total and distinct behaviours evaluation 39
4.5 Monkey vs DriveDroid: total and distinct behaviours evaluation 40

Chapter 1

Introduction

The growth of Android malware poses a serious threat to mobile systems. With a current market share of over 82.8% (2015 Q2 [1]), Android is dominating the mobile scene with both the official Google Play Store and several third party stores. Malware authors can bypass Google security checks (e.g., Bouncer [2]) by just repackaging popular applications with malicious code and publishing them in these alternative markets, contributing to the overall spreading of malware. As stated in [3]: "With strong market share, and with an open, user-friendly platform, the Android OS remains a top target for malware;" this has been witnessed over the last few years.

Malware analysis is therefore a relevant research field, with both static and dynamic analysis techniques. Static analysis allows to examine a program without executing it, with the known advantage of ample code coverage. However, it is strongly limited by obfuscation techniques and dynamic payloads. With dynamic analysis, the malware is actually executed and the analysis aims to understand what is happening while the malware is running in an emulated environment. This approach overcomes static analysis limitations: at some point the malware has to deobfuscate itself and, since the program is running, it is possible to analyse dynamic payloads. However, even this technique has limitations. First, it can only explore executed paths: if malicious behaviour is not in an executed path, it will never be exhibited. Moreover, modern malware purposely hide malicious behaviours if they discover, through the use of sophisticated interaction challenges, they are under analysis. These techniques are popular in the non-mobile malware world, and are expected to be ported to mobile malware as well, as it happened for other evasion techniques in the past. Therefore, one of the dynamic analysis main challenges is finding a way to stimulate an application, triggering as many malicious behaviours as possible, mimicking the activity of a real, human user.

Stimulating an application is not a trivial task: current solutions exploit stress-test tools such as *Monkey* [4] or combine static and dynamic analysis. The main problem with these solutions is that they either cannot reach a satisfactory code coverage or cannot always reproduce a typical human interaction, giving the assumption that applications are meant to be easily used by humans in all their functionality. A recent work, PuppetDroid [5], tries to overcome this limitation: the idea is letting the user stimulate an application and to reproduce the interaction on other applications considered similar. Two applications are considered similar from a user interface (UI) point of view. The main limitation of PuppetDroid is the lack of a model for recording and rerunning an application: they simply record where the user interacts on the screen and then reproduce the same interaction. This can lead to dead ends, where no interactions are available or, worse, to application crashing because of wrong inputs.

In our work, we propose DriveDroid, an extension to PuppetDroid, that models the interactions a human performs to exercise an application. Recalling the PuppetDroid intuition, we let users stimulate applications and we record their interactions. In addition, we represent the interactions as a directed graph where the nodes are the views (i.e., screens) of the application and the edges are the actions a user can take from each view. Further on, we design a novel strategy to use, so to "amplify" the exploration capabilities of the human user who provided the UI interactions. Finally, we employ a UI similarity-based approach in order to cluster applications; the resulting clustering is queried to lookup a stimulation graph whenever a new application needs to be analysed.

In order to validate our approach, we conducted experiments on a dataset of 1,028 Android applications. In particular, we showed that DriveDroid can trigger at least as many behaviours as the state-of-the-art approaches (*Section* 4.3). Moreover, we showed how our approach can deal with extreme UI-based evasion techniques (*Section* 4.4). Finally, our results show that DriveDroid on our modest dataset can reuse traces recorded by human users with a factor of 5 on average (up to 30 in the best case). This means that DriveDroid can effectively multiply the effort of the human analyst. With these results, the idea of creating a crowd-based, user-centric dynamic malware-analysis system is very promising and feasible as an extension of our work.

In summary, this thesis makes the following contributions:

- We provide a new model and framework for stimulating Android applications.
- We provide a way to face extreme user-interaction-based evasion techniques, which are not handled by state-of-the-art, test-automation mechanisms.
- We provide a framework that can reduce the effort of a malware analyst, while performing sound stimulation.

Chapter 2

Background and motivation

In this chapter we give an overview on the Android platform, the state of the art related to our work and the motivation for our work. In *Section* 2.1 we briefly speak about Android and its diffusion. In *Section* 2.1.1 we explain the principles behind the Android user interface and their implementation. In *Section* 2.1.2 we give a high level overview of the `MotionEvents` system used by Android to manage UI events. In *Section* 2.1.3 we speak Android malware and the principal analysis techniques employed against them. In *Section* 2.2 and *Section* 2.2.1 we explain the state-of-the-art approaches to automatically interact with Android applications UI. In *Section* 2.2.2 we define the concept of *interactive challenge* and we show its importance in malware analysis and an example of its implementation. In *Section* 2.2.3 we speak about the state-of-the-art limitations. Finally, in *Section* 2.3 we explain the main goals of our work and the challenges we had to face in its development.

2.1 Android

Android is an open source operating system (OS) designed to run on a large variety of smartphones. Its open nature and its easy customization makes it very appetible for many different vendors as the default OS for their phones. For this reason Android is the most common software solution in the mobile market, with a share of over 82.8% (2015 Q2 [1]). The Android-compatible devices are build up by a lot of different companies, and, thus, they come with different characteristics (the so called fragmentation), both in their design and at performance level. Moreover, many different versions of Android are active at the same time, introducing another level of fragmentation [6].

In *Sections* 2.1.1 and 2.1.2, we provide a brief explanation of how the UI is implemented and how Android deals with the events injected into the device.

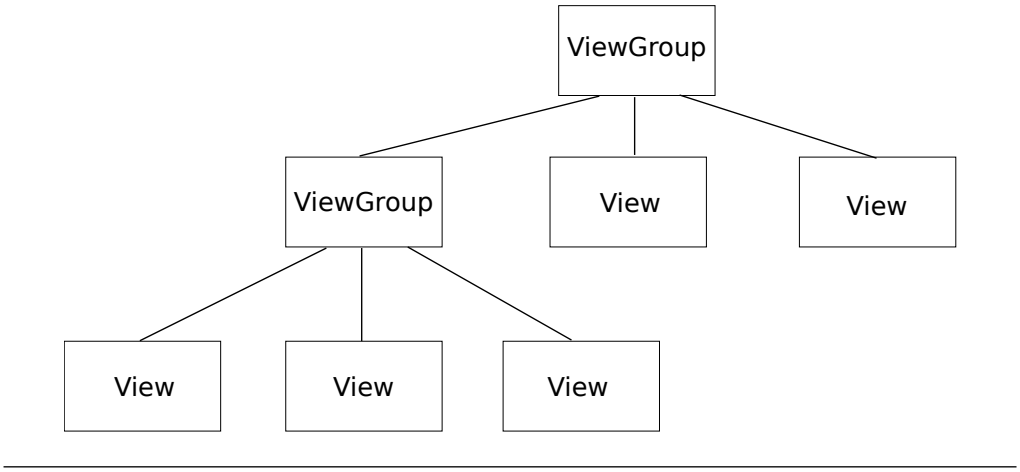


FIGURE 2.1: Android abstract views hierarchy

In *Section* 2.1.3, we give an overview of Android malware and of some analysts works which aim rendering easier the malicious apps recognition and the countermeasures.

2.1.1 Android user interface

The Android user interface model is a hierarchical structure where each component is a container for the elements of the layer immediately below: the developer can create the UI tree by defining and nesting various elements by means of XML files. As depicted in *Figure* 2.1 and *Figure* 2.2, usually the root node is a `ViewGroup` which is a meta-element whose only purpose is to contain other `ViewGroups` or `Views` which hold the actually displayed content. Both `ViewGroups` and `Views` are generic `Containers` which are extended to draw the actual UI. The most common implementation of `ViewGroups` are the `Layouts`: they are nodes which may or may not contain other sub-layouts. Their role is to create a sort of "logical grid" where all the other components are placed with a disposition strongly dependent on the chosen `Layout`. At some point a `Layout` can contain other "abstract" elements whose role is to arrange the child elements positions (i.e., `TableRows` and `ListViews`). `Views` are the leaf objects that are actually displayed: `Buttons`, `TextViews` (elements with written text), `EditText` (elements where the user can write text), etc. The hierarchical structure is (directly or indirectly, through the use of `Fragments`) contained in one or more `Activities`, which are the application elements where the UI and the business logic interact.

Listing 2.1 shows a practical example, taken from [7]. Note that part of the semantics can be expressed dynamically directly in the code, calling and customizing the XML declared elements with the API provided by Android. Therefore, an exhaustive lookup of all the UI elements by parsing the XML files would only find statically defined elements.

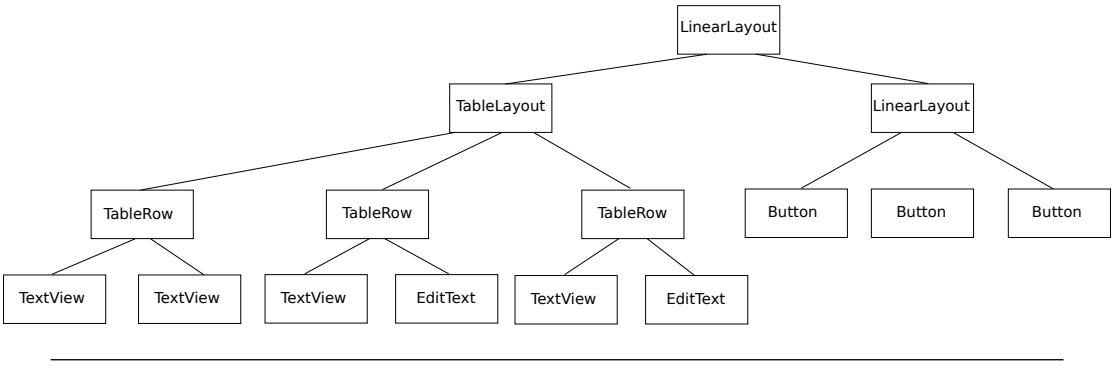


FIGURE 2.2: Android detailed views hierarchy

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:layout_width="fill_parent"
              android:layout_height="fill_parent"
              android:orientation="vertical" >
    <TextView android:id="@+id/text"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:text="I am a TextView" />
    <Button android:id="@+id/button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="I am a Button" />
</LinearLayout>
```

LISTING 2.1: XML Structure of a view: a simple layout with text and a button

While an application is running it is possible to interact with its currently displayed UI elements as well as retrieve the complete tree structure, including dynamically defined elements. In our work, as described in *Section* 3.2.2, we use this layout information to define the UI states. One of the most useful tools to test an application UI is *UI-Automator* [8], which allows to scan and analyse the UI components currently displayed by the running application: for each element it is possible to obtain position, size, class, contained text and other UI-related values. UI-Automator can also be used to inject events to the View elements at runtime as if a human were interacting with the application.

2.1.2 Android events management

In Android the events are represented at source code level by `MotionEvents`: this representation wraps the type of the event (Touch Up, Touch Down, Key Pressed, etc.) and the coordinates where it happened along with event-specific parameters (for instance, the key pressed in a KeyEvent). When an event occurs, the OS translates the corresponding `MotionEvent` to a lower level representation and passes it to the respective "manager".

For our purposes, to extract events information from the device drivers and reconstruct from that the high level representation, it is sufficient to know the `MotionEvent` representation, since for any further level of abstraction it is possible to rely on "getevent," a tool running on the system and callable by ADB.

2.1.3 Android malware

Android popularity and its relatively open ecosystem have facilitated the growing number of malware targeting it. The malware authors leverage social engineering tactics to get more permission than needed. Most Android malicious apps are repackaged versions of legitimate applications. They will therefore have the same UI and functionality of the original, plus some malicious code.

Not only the overall amount of Android malware is remarkable, but the newly discovered malicious applications number (about 4,900 new Android malware every day [9]) is far beyond the reasonable amount that analysts can handle without automating the analysis procedure. Therefore, two main techniques are normally employed: static and dynamic analysis. The former inspects the code without executing the application, thus gaining good code coverage, although it is useless against code obfuscation techniques. The latter executes the application in an instrumented and sandboxed environment. Automatic stimulation techniques can be used to let the app navigate through the views and explore as many code path as possible. However, malware have become sophisticated and able to recognise emulated environments (which can be mitigated refining the emulators), or resort to "interactive challenges," which an automatic tool is very likely to fail. When a malicious application notices either an emulated environment or a non-human user, it might halt its execution or, worse, prevent the triggering of some very specific behaviours, rendering dynamic analysis potentially harmless for it. In *Section* 2.2.2 we describe in details the concept of "interactive challenge" using an application that we created for the purpose of demonstrating what a malware author may employ to easily evade naïve UI-stimulation approaches.

2.2 Stimulation of Android applications

The normal way for a user to interact with an Android application is by performing actions semantically related to the UI element displayed. When it comes to design a strategy for an automated system to interact with an application, a number of problem arise:

Syscall	Target	Additional info
open	/sdcard/myfile	`READ_MODE`
read	/sdcard/myfile	
...	...	...
close	/sdcard/myfile	

TABLE 2.1: Behaviour example: notice the system calls.

- Dynamically declared elements can be retrieved only by tracking the UI during execution, which is not efficient; there is, therefore, a trade off between code coverage and efficiency.
- Even assuming complete knowledge of all the UI elements, it is not trivial to chose an appropriate action, especially if the UI elements involve writing text and clicking on a sequence of buttons, for instance.

From the technological point of view, it is not hard to (blindly) automate a UI stimulation, due to tools like UI-Automator, which is capable to inject events and provide information on some specific values or states of the app, for instance, the screen orientation (portrait or landscape) and size. An important remark is that UI-Automator does not provide a strategy to stimulate an application: it just provides an API to the stimulation tool, so the strategy is up to the user, human or not.

Regardless the approach employed by malware analysts, the goal of exercising an Android application is stimulating behaviours. A behaviour is a set of system calls logically mergeable together, building a high-level semantic. For example, a behaviour such as sending an SMS is translated to the invocation of two `ioctl` system calls on `/dev/binder`: a call to locate the service and the other to invoke its method. Other examples of behaviours are: network access (e.g., DNS and HTTP requests, etc.), access to personal info (e.g., location, contacts, phone info and SMS), execution of an external application, call placing and file system access. *Table* 2.1 shows the system calls trace for a behaviour (the application reads the external storage).

2.2.1 Exercising Android applications

When it comes into play to select a strategy for stimulating an application, two are the main possible approaches: a random stimulation or a non-random stimulation.

The random approach is the easiest one: it consists in selecting a random event from all the possible ones and injecting it to a random position, then repeat the process for an indefinite number of times. It is also possible to partially refine the approach by

not selecting at all (or with a very low probability) events which are likely to draw the execution outside the context of the running application, wasting this way one or more successive inputs. The main state-of-the-art implementation of the random approach is Monkey, a tool capable of generating a variable number of random inputs and injecting them into a running application at the desired speed. It is provided with the Android SDK and it is launchable from ADB; the tester can specify the number of events to be injected, as well as the eventual pauses between them, the package name of the app to target and other minor parameters (see [4]). Its main advantages are that it is totally automatic, easy to use and it can inject a remarkable number of events (some thousands) in a relatively short time (a few minutes).

Monkey benefits are maximized when it is embedded in a more complex framework which demands to an oracle (for instance, static analysis) the decision on when and how running it on an application. One of the most notable state-of-the-art example of an hybrid-random approach is *Andrubis* [10], which is, at the best of our knowledge, the biggest public available analysis system for Android applications. In a first step Andrubis leverages static analysis by parsing the manifest file of the application and some functions from the byte code. In this way, Andrubis is aware of the main structure of the application, knowing all the activities and some other components such as services, broadcast receivers etc. This information is used to exercise the application UI (dynamic analysis) leveraging Monkey; this is done for all the action based UI component such as buttons, text input, etc.

Another work, *DynoDroid* [11], leverages a random approach. Differently from Andrubis, DynoDroid uses *MonkeyRunner*, an Android SDK tool providing an API for writing programs that control an Android device or emulator from outside of Android code. The main idea behind DynoDroid is to trigger only the relevant events for the application; after an event is triggered, an observer computed if this event is relevant or not (it is relevant if it may result in executing code that is part of the app) and add it to the set of all the relevant events. A selector selects from the relevant events, using a random strategy, the event to trigger which is injected by an executor.

Opposed to the random approach, we have another class of stimulation techniques: all of them have in common a non-random way to generate the events to inject to a running application. The generation techniques are dependent from the approach and its implementation, but they have in common the fact to output a considerably lower number of events.

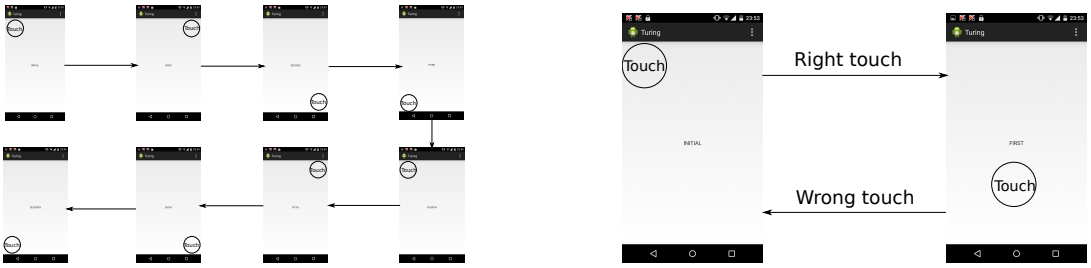
Most of the proposed approaches combine static and dynamic analysis in order to improve the quality of the stimulation. One of the most interesting works in this sense is surely *SmartDroid* [12]. In a first phase, SmartDroid leverage static analysis in order to

generate a function call graph (FCG) and an activity call graph (ACG). With the FCG all the function calls leading to a sensitive API are found while the AGC defines the activity related to the sensitive function call, defining it as a sink activity. In a second phase SmartDroid performs a dynamic analysis implementing an automatic approach to validate the path previously found. One of the most recent works not using a random approach is *Andlantis* [13]. Andlantis is a large scale architecture aiming to dynamic analyse Android applications. To explore the UI of an application, Andlantis uses MonkeyRunner, aiming to explore as many features as possible: in case a state has no more UI elements to visit, the previous state is reached and the exploration restarts from there. Finally, we want to cite *CopperDroid* [14], a system capable of reconstructing Android malware behaviours. CopperDroid tracks system calls and by an unmarshalling procedure, it is able to reconstruct complex Android objects and extracting behaviours from them. The author intuition is that all behaviors are eventually achieved through the invocation of system calls. To exercise an application an then reconstructing behaviours, CopperDroid leverage static information extracted from the application to perform a later stimulation. In particular, it examines the application manifest, extracting events and permission-related information to drive the stimulation performed by means of MonkeyRunner.

2.2.2 Motivating example

In this section we describe the concept of *interactive challenge* using an application that we created as an example for demonstrating what a malware author may employ to easily evade naïve UI-stimulation approaches. We define a interactive challenge as a non-trivial, very precise stimulation sequence required by an application in order to change its UI state or to take some code paths unreachable otherwise.

In desktop malware a common sandbox-analysis-evasion technique is to perform the malicious behaviours after some very specific actions have been performed on the user side. Examples are the "click here" game and a CAPTCHA-like form to fill. This has been proved an effective way to hinder dynamic analysis and we believe there is the possibility that in the future such technique may be ported to mobile systems, including Android. We are confident about this due to the event-driven nature of Android applications, where a user stimulation is needed to trigger the majority of the events: it would be easy to hide an interactive challenge in the very first view of the application and simply do nothing (or even perform only legitimate behaviours) if no proof of a human user are recorded by the malware itself.



(A) Right interaction leading to hidden behaviours.

(B) Wrong interaction leading to start from the beginning.

FIGURE 2.3: Interactive challenge: right and wrong stimulation of our application.

We developed an app employing this evasion technique to explain the concept and to show that a random stimulation approach cannot easily cope with evasion mechanisms based on interactive challenges in general. *Figure* 2.3a shows the correct stimulation flow; in *Figure* 2.3b is depicted the effect of any wrong interaction. After every step a `TextField` informs the user about the state of the stimulation. Whenever a tap is performed in the wrong place/order, the state is reset to the initial one. Only after the sequence has been performed correctly, the application shows its true malicious behaviours. The corners, which are the correct spots to tap, are defined as the area between the screen upper (or lower) border, the screen left (or right) border, the first quarter of the "short" screen dimension (in our case the width) and the first sixth of the long screen dimension (in our case the height). This means at every tap a random approach (i.e., Monkey) has probability 1/24 to tap in the right place. Since 8 taps in 8 places must be done in the right order, the final probability is $(1/24)^8$ which is less than $(1/2)^{32}$, which is more or less one over 4 billions, all assuming that the random stimulation injects only taps.

2.2.3 Limitations

In *Section* 2.2, we explored some of the proposed approaches to increase the code coverage of dynamic analysis. However, random solutions suffer of an evident limitation: the random approaches on average require a very long time and an extremely high number of injected events to gain an ample code coverage, which is not worth expecting. Moreover, for random solutions such as Andrubis or DynoDroid, even a long execution time is useless: the lower the time between the events injected is, the higher the probability the application might crash due to "wrong" inputs or pre-existent bugs is: this is deleterious for a random approach, because in the best cases it just stops or remains stuck and in the worst scenario it continues the stimulation out of the application context, essentially wasting time. In particular the exiting-from-context problem is not only limited to the application crashing, but it may also be a consequence of the "back" event injected when the app has not a valid previous view, in which case it is simply put into background

execution; this would not be a problem if it were possible to detect this happening and counter-react with an apposite sequence of events bringing the application on foreground again. Other solutions try guide dynamic analysis with a preliminary static analysis. The main problem with these techniques is that they suffer of the static analysis limitations: in case of code obfuscation or packed applications, those solutions will not work. Finally there are some limitations common to all these approaches: input forms and interaction tests. The first category is (almost always) a dead end for every kind of automatic stimulation approaches, especially when a login is involved. The second category involves completing a non-trivial sequence of actions in a certain (non predictable for non-humans) order to be able to proceed. This last category is widely employed in desktop malware to detect and evade sandboxed analysis environments and automatic stimulations, and, although there is no on-the-wild Android implementation of a similar concept yet, it is likely this technique will be ported to mobile malware in the close future.

2.3 Goals and challenges

Having considered the limitations of the state-of-the-art approaches summarised in this section, we designed DriveDroid with two main goals in mind:

1. To design a new approach to stimulate Android applications by recording and rerunning user interactions, to overcome the interaction-based evasion techniques while maintaining good code coverage.
2. To implement the approach in a sound way, in order to create a substrate for a future easy-to-use and crowd-sourcing public system.

To achieve the first goal, our idea is to create a model representing in a novel way the states of the UI, the interactions needed for transitions and the event to inject in order to make the transition happening. To achieve the second goal, our idea is to re-engineer the existing PuppetDroid infrastructure from scratch.
Moreover the system performances must be appropriate for a large-scale deployment. For these reasons we foresee different challenges.

The first challenge is to build a sound but easily usable model, without overfitting (e.g., too many states, each one differing from the other(s) by insignificant details) and without affecting the system performances. In particular the representation must be expressive enough to allow the re-execution phase to work and compact enough to allow a simple parsing for performance reasons. The challenge arises because these two requirements

have the exactly opposite solutions (in the first case, adding more information, in the second case, abstaining from loading the graph with additional data).

The second challenge is the re-execution: we must avoid the process to remain stuck, or, at least to detect the dead ends and react to them as soon as possible. This means the system has to constantly check whether the re-execution is in a stuck state or in any other sort of trap which prevents the process to go on. The challenging part in this is ensuring a quick and sound solution for this problem, even at the cost of a dynamical modification of the model during the re-execution. This means that also the model has a further requirement: the runtime modification capabilities, which adds complexity as well.

The third challenge is to make DriveDroid a distributed system working in a multi-user scenario, where multiple requests at the same time can be directed to the components. The challenging parts in this are designing the system with in mind a multi-threading implementation, interact in a sound way with the Android SDK tools, and correctly integrate all the components which run on different machines. This is the most problematic point: the Android SDK tools are known for being far from bug-free and the communication between different machines must satisfy some minimum security requirements (confidentiality, integrity, authentication).

Chapter 3

DriveDroid

Here, we present how we redesigned PuppetDroid, starting with an explanation of the new approach we used (*Section* 3.1), moving up with the explanation of how we implemented it, describing in details every component needed for a sound functioning (*Section* 3.2 and *Section* 3.3).

3.1 Approach overview

We propose a new approach based on modelling the user interactions with an Android application as a directed graph where the nodes are the views and the edges are the actions (user gestures) performed by the user to navigate the application. We use this model in four different phases: in **Phase 1 (Clustering)** we cluster applications according to their UI similarity; in **Phase 2 (Recording)** we record the user interaction and generate the aforementioned m̃odel; in **Phase 3 (UI-similarity graph lookup)**, when a new application needs to be executed, a graph model is looked up by querying the clusters obtained in **Phase 1 (Clustering)** according to the UI similarity, and in **Phase 4 (Re-Execution)** the graph is used to rerun the new application. The final result is an Android UI exerciser able to trigger behaviours, sets of system calls logically merged together building an high level semantic (see *Section* 2.2). In the following sections we explain in detail all the phases mentioned above.

3.1.1 Phase 1: Clustering

In this phase, we aim to obtain clusters of similar applications. As for every type of clustering, the most important and challenging part is defining a similarity (or distance) function between the points in the dataset. We consider two applications to be similar

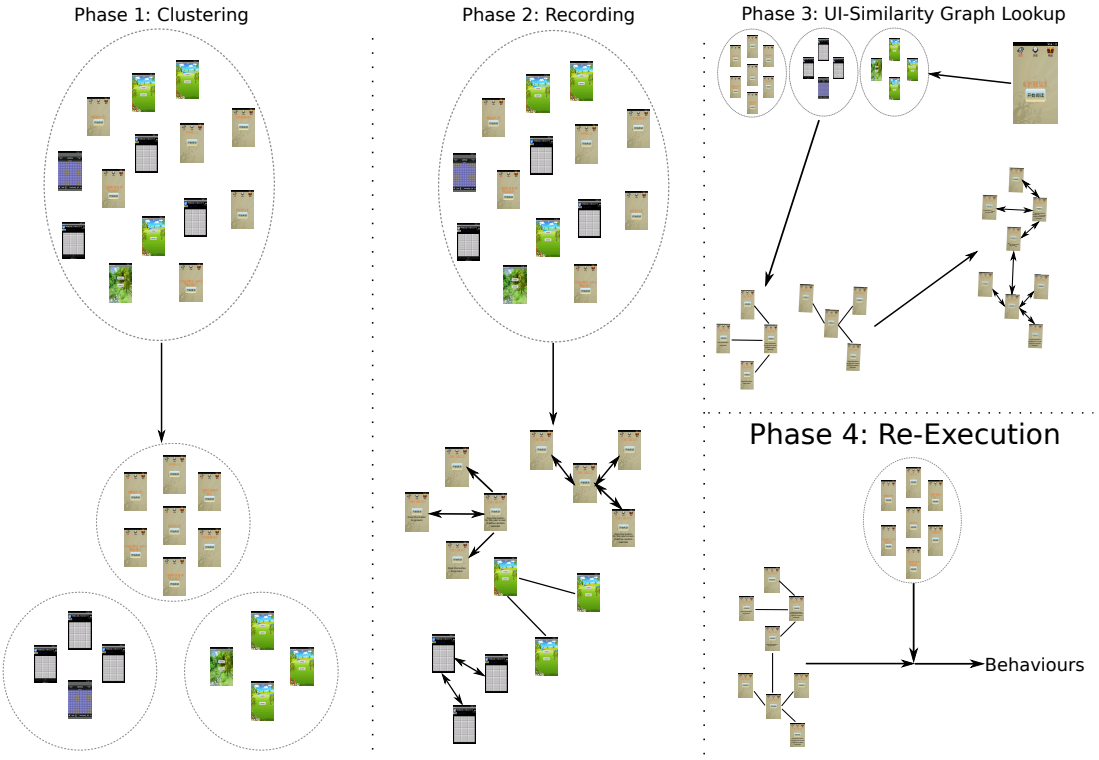


FIGURE 3.1: This picture shows, from an high level, our approach phases. In **Phase 1** we cluster applications, in **Phase 2** we record graphs, in **Phase 3** we lookup and merge graphs from the clusters and in **Phase 4** we re-execute the merged graph on target application.

if they have a similar user interface. We informally define the *similarity* between two applications as the degree of visual similarity of their respective views. We relax this definition by considering two applications as "similar" if their first views (i.e., at boot) are visually similar. Although comparing more than one view per application may provide more accuracy, we discard this option in favour of efficiency.

Given this similarity definition, which will be better defined in *Section* 3.2.2, this phase clusters applications using a hierarchical clustering algorithm. The output is thus a set of clusters containing UI-similar applications. For each cluster, we elect a representative point, which is essentially a centroid. These centroids are used in **Phase 3 (UI-similarity graph lookup)** to speedup the lookup of a cluster of applications which are similar to the given one.

3.1.2 Phase 2: Recording

In our graph-based model, which we will formally define in *Section* 3.2.2, recording an interaction means binding each view of the application explored to a state, and each user gesture to an edge.

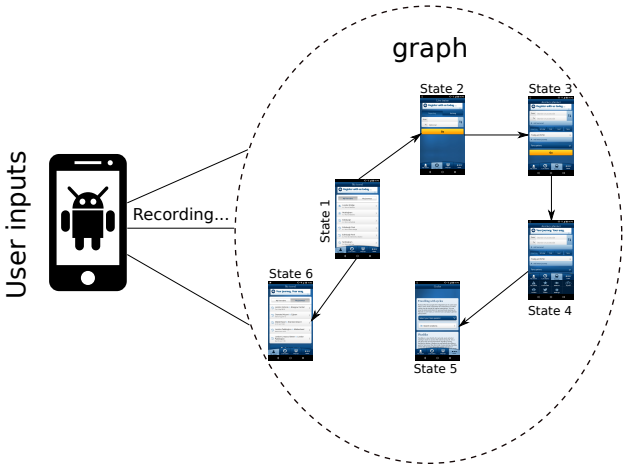


FIGURE 3.2: Graph recorded after the application went from state 1 to state 5 passing through states 2, 3 and 4.

For example considering an application with 6 views, a possible user interaction could be to navigate through the states from 1 to 5. In this case, if the user decides to interrupt the test, state 6 will never be reached and we would miss some possible actions to add to the graph which would result without those states and edges. The resulting graph is depicted in *Figure* 3.2.

Multiple users can stimulate the very same application: different graphs of the same application can be found. We perform a merge of all the graphs found, generating a super-graph, obtaining it in this way:

- We perform the set union of the nodes.
- When merging two nodes, we perform the set union of the edges exiting from them.

Moreover, while recording, we keep track of how many times a user performs an action. This way, we can associate a frequency to each edge.

3.1.3 Phase 3: UI-similarity graph lookup

To lookup a previously recorded graph, we leverage the definition of similarity explained in *Section* 3.1.1 The target application is launched and from its first view we select the cluster that best fits the application. From this cluster is then possible to retrieve the graph to rerun on the target application (see *Figure* 3.3).

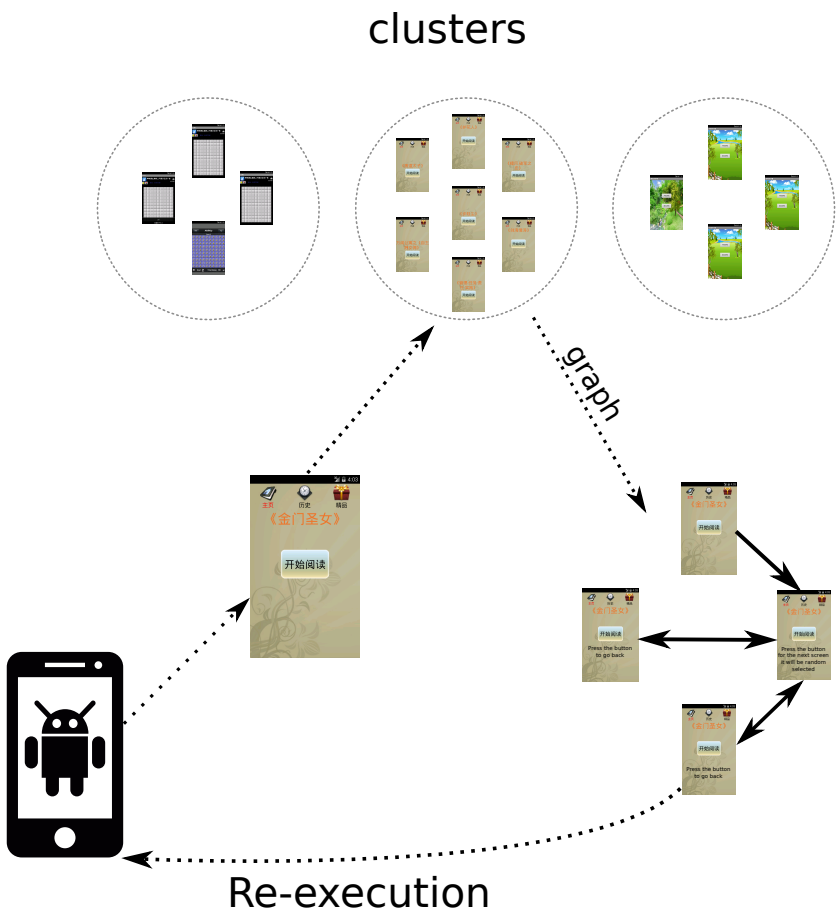


FIGURE 3.3: We take the screenshot of the first view shown and assign it to the respective cluster. From this cluster we retrieve the merged-graph to re-execute.

3.1.4 Phase 4: Re-execution

To re-execute a graph, we leverage the frequencies attached to each edge, which are essentially the empirical probabilities that the users take that edge. There are nodes without outgoing edges, because a user can stop an interaction at any time; we call these nodes *"stuck nodes."* In principle, we could detect and eliminate all the stuck nodes statically before re-executing the application. However, in this way we will never reach that node, loosing some possible behaviours.

For this reason, we do this operation dynamically and only when the stuck node is explored. To do so, we restart the application from the beginning since backtracking may lead the application to an inconsistent state, possibly causing a crash. Before the application is restarted, we eliminate all the edges leading to that node. This way, we force the execution exploring different nodes, enhancing the probability of triggering more distinct behaviours. It is possible that, when the application is restarted, the initial

FIGURE 3.4: State 2 is a stuck node. When reached the edge from state 1 to state 2 is removed and the application restarted. The application will restart from state 1 which now is a stuck node.

node becomes a stuck node: in this case restarting the application is not useful, since we will finish in the very same stuck node; a simple example is provided in *Figure* 3.4.

To face this problem, we take a screenshot of the last node we reached while re-executing. We use this screenshot to lookup a graph in the same way done for **Phase 3 (Ui-similarity graph lookup** (*Section* 3.1.3). In this way, we keep exercising the application, with the chance of exploring new views and triggering new behaviours. In conclusion, due to how we implemented the exploration of the graph to avoid stuck nodes, this phase results in the use of multiple graphs.

3.2 Implementation details

In this section we show how we implemented our approach. In particular, in *Section* 3.2.1 we explain how we implemented **Phase 1 (Clustering)** and **Phase 3 (UI-similarity graph lookup)** and in *Section* 3.2.2, we give a more concrete idea of what a graph is and how we implemented **Phase 2 (Recording)** and **Phase 4 (Re-execution)**.

3.2.1 Similarity and lookup

In this section we give a deeper explanation of the algorithm we use for clustering and the way we perform a lookup.

Choosing a metric of distance between application is essential. To this end, we leverage the concept of *perceptual hash* (pHash). According to [15], a pHash is a fingerprint of a multimedia file derived from various features of its content and it is robust to distortions, rotations, etc. Moreover, differently from a common hash used in cryptography, where a slight change in the input result in very different output, the pHashes are close to each other if the features (input) are similar. Since we are comparing images, we exploit this property to derive a distance function for our clustering algorithm based on the hamming distance.

Phase 1: Clustering

To cluster applications, we adopted an hierarchical clustering algorithm. Despite it is not the fastest clustering algorithm, we choose it since it best fits to our problem, as for now we only have a distance function (as opposed to a mapping of the image features to a metric space).

There are two ways for hierarchical clustering: agglomerative or divisive. We implemented an agglomerative strategy because for us it was the easiest to implement, and it allowed us to tune the algorithm in one run, exploiting the different levels of the dendrogram: we start considering all the applications as a single cluster, merging the closest clusters step by step, until we have just one big cluster. The result is a dendrogram on which we then tune the algorithm and to choose the best cut. In order to decide the best cut, we evaluate our algorithm with an knee-elbow technique [16]. This technique, looks at a proper defined index as a function of the number of clusters: we choose the number of clusters so that adding a new cluster to that number will not give any advantage; as we will show in *Section* 4.1, we used a between and within sum of squares as indexes.

After the clusters are generated, for each cluster we elect its barycenter as the representative point, which serves the purpose of a centroid. To calculate the barycenter of a cluster, we compute the distance of an image from all the other images in the cluster and we sum the results. We repeat this procedure for each image and we return the image with the minimum sum of distances.

Phase 3: UI-similarity graph lookup

Before and while re-executing an application ("target application"), we need to lookup a graph corresponding to a similar application. Thanks to the centroids that we derived for each cluster, looking up the cluster that best fits a target application (given its UI) is fast. More precisely, this approach is one order of magnitude faster than a naïve lookup that spans across the entire dataset, as we show in *Section* 4.2.3. Once we selected the best cluster, we have to retrieve a graph and start the **Phase 4 (Re-execution)**. Defining an ordering between graphs it is not possible, since they are recorded on not completely similar applications. In principle, we could select the graph recorded on the application which is the most similar to the new one. However, this operation will nullify the performance gain obtained with the centroids. For these reasons, we decided to adopt a random strategy: we choose randomly from the graph available in that cluster.

State	**Comments**
Interaction list	List of the user interactions: swipe, touch...
Encountered	How many time the state has been encountered
Duration	How long the user stayed in this state
UI Elements	graphical elements in the screen

TABLE 3.1: State implementation

3.2.2 DriveDroid graph

From an high level point of view, an Android application is a set of views (screens) containing elements such as buttons, text views, text fields etc. When exercising an application, a user interacts with all these component performing actions that allow him/her to navigate through different screens. Therefore, a state in the graph has to take into account which graphics elements are present in the screen as well as which actions the user performs on that screen.

Table 3.1 describes what a state is in our implementation. In each state, we keep a list of interactions which are objects representing a user action and wrapping all the information useful to trigger it when requests (i.e., UI element to exercise, frequency of the interaction etc.); encountered is a variable useful during the re-execution for computing the empirical probability of an edge to be taken; duration is a variable which value represents how long an user stayed in that state, it is essential to avoid too fast interactions and to better mimic the user gestures; Finally the UI root element is the root element in the tree hierarchy of Android UI (see *Section* 2.1.1); a graph is then a set of states of the type just described.

Phase 2: Recording

This phase consists in recording and encoding the user interactions as a graph. Technically, in this phase we use the Android emulator capabilities to dump the components of the current view of a running application and read the user gestures that are injected in the application. For the purpose of this work, an *event* is defined as an atomic action performed by the user on the screen: `TouchUp` and `TouchDown`. Moreover, an *interaction* is defined as a set of gesture made by the user from the first `TouchDown` to the last `TouchUp`; examples of interactions are touch, swipe and long touch.

In *Listing* 3.1, we show a pseudo-code of the recording procedure. Once the application starts running, we capture the initial state and we add it to the graph. Then we wait for new events of the type explained above. When a touch up occurs, we start recording a new state and when a touch down is performed, the state is finally added to the graph and the process goes on until the user decide to disconnect from the recording session.

```
def record():
   graph = new Graph()
   new_state = get_initial_state() #create a state wrapping the first screen of the application
   last_state = None
   interaction = None
   time = get_time()

   while phone is connected:
      event = read_event() #read a touch up or touch down event from the device

      if event is TOUCH_DOWN:
         graph.add_state(new_state)

         if last_state is not None:
            interaction.final_state = new_state
            last_state.add(interaction)

         last_state = new_state

         if interaction is not None:
            interaction.set_duration(time - get_time())

      if event is TOUCH_UP:
         interaction = Interaction() #create a new instance of the interaction object
         interaction.set_involved_element()
         new_state = create_state()
         time = get_time()
```

LISTING 3.1: Recording graph pseudo-code

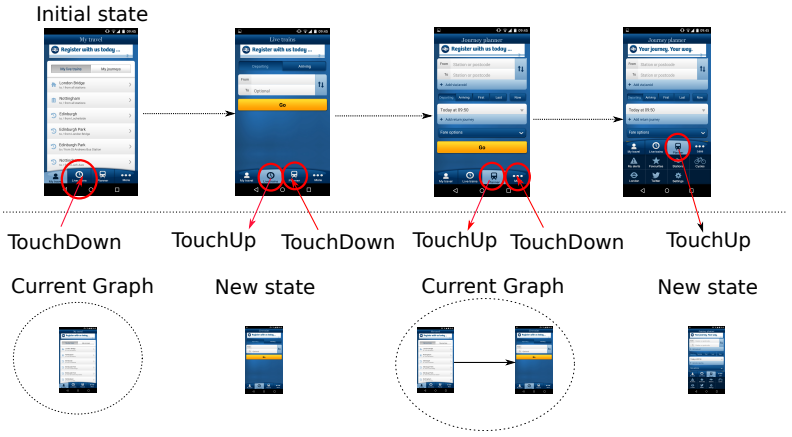


FIGURE 3.5: The initial state is immediately added to the graph and after a touch down and touch up a new screen is reached. Afterwards, another touch down is performed: the screen change and the after a touch up a new state is created. Finally A new touch down occurs an the new state is added to the graph while another new state is generated.

In *Figure* 3.5, we show a graphical representation of the main passages of this phase.

Phase 4: Re-execution

Once the graph is recorded, we need to explore it, re-executing the interactions over another similar application; *Listing* 3.2 shows the pseudo-code for the re-execution.

```
def rerun(graph):
   current_state = graph.get_initial_state() #The current state is set to be the initial one
   encountered_states_ids = []

   while True:
      encountered_states_ids.append(current_state.state_id)
      #getting interactions list
      interactions = current_state.get_interactions()

      #Relaunching part in case of a stuck state
      if interactions is Empty:
         from the graph, remove interactions leading to current_state

         #remove current state from the graph
         graph.remove(current_state)

         stop_and_relaunch_application()
         current_state = graph.get_initial_state()
         interactions = current_state.get_interactions()

         #Check if we are stuck again after restarting
         if interactions is Empty:
            notify for a graph switch


      chosen_interaction = select_interaction(interactions)
      fix_interaction_position(chosen_interaction)
      inject_interaction(chosen_interaction)
      chosen_interaction.is_taken(True)
      current_state = chosen_interaction.final_state
```

LISTING 3.2: Exploring graph pseudo-code

Binding each node to the current view shown is fundamental for re-executing a graph over an Android application. Therefore, each edge is an interaction that can be injected directly into the Android emulator where the application is running: when we transition from a node to another in the graph, the application changes view in turn, keeping all synchronized. We exploited a probabilistic approach to choose the interaction to perform: each interaction has a probability of being chosen that depends on how many time the user performed it during **Phase 2 (Recording)**. This strategy has been implemented in the method `select_interaction` used in the pseudo-code shown in *Listing* 3.2. During the re-execution, it is possible to end up in a "stuck node": in this situation we restart the application from the beginning, and we remove all the interactions leading to that node from the graph. However, removing all the interactions leading to a stuck node could force other nodes to be of the stuck type: restarting the application could bring to a stuck state at the beginning, compromising any type of re-execution. To face this problem we take a screenshot of the last node we reached while re-executing; we use this screenshot to lookup a new graph to re-execute, starting from the last view reached; in *Listing* 3.3 we show a pseudo-code of the switch procedure.

```
1 def switch_graph():
2    screenshot = take_screenshot()
3    cluster = get_best_fit_cluster(screenshot)
4    graph = lookup_for_grah(cluster)
5    rerun(graph)
```

LISTING 3.3: Performing a graph switch

3.3 System architecture overview

In order to validate our approach, we implemented it in DRIVEDROID, extending and redesigning the old PUPPETDROID. The clients can search for APKs and start a test session by sending requests to a *Main Server* which dispatches the jobs to a *Worker*.

The Main Server has the role of dispatching the requests from the *PuppetApplication* (the client) to the Worker. In this way the system is more scalable and future modification or extensions are easier. The routine to initialise a recording session goes through the Main Server: from the PuppetApplication the user looks for an application to test sending the query to the Main Server; if the application is found the user can select it for a test session an forward the request to the Main Server that in turn communicate with the Worker in order to initialise the test environment. Finally the Worker and the PuppetApplication are connected and the test begins.

In the following sections we give a detailed explanation of the two core component of the system: the PuppetApplication and the Worker.

3.3.1 PuppetApplication

The *PuppetApplication* is installed on every client as part of the implementation of **Phase 2 (Recording)**. It is an enhanced version of the PUPPETDROID client with a new functionality (registration, see *Section* 3.4.8) and significant refinements to make it compatible with the newest Android technologies, versions and emulators. After a mandatory registration, the application offers two main functionalities: search for an application to test (*Section* 3.4.8) and perform a test on the selected application (*Section* 3.4.8).

Since the target application to test is not running directly on the user phone, the PuppetApplication allows the user to interact on the remote emulator giving him/her the feeling the application is actually executed on his/her phone.

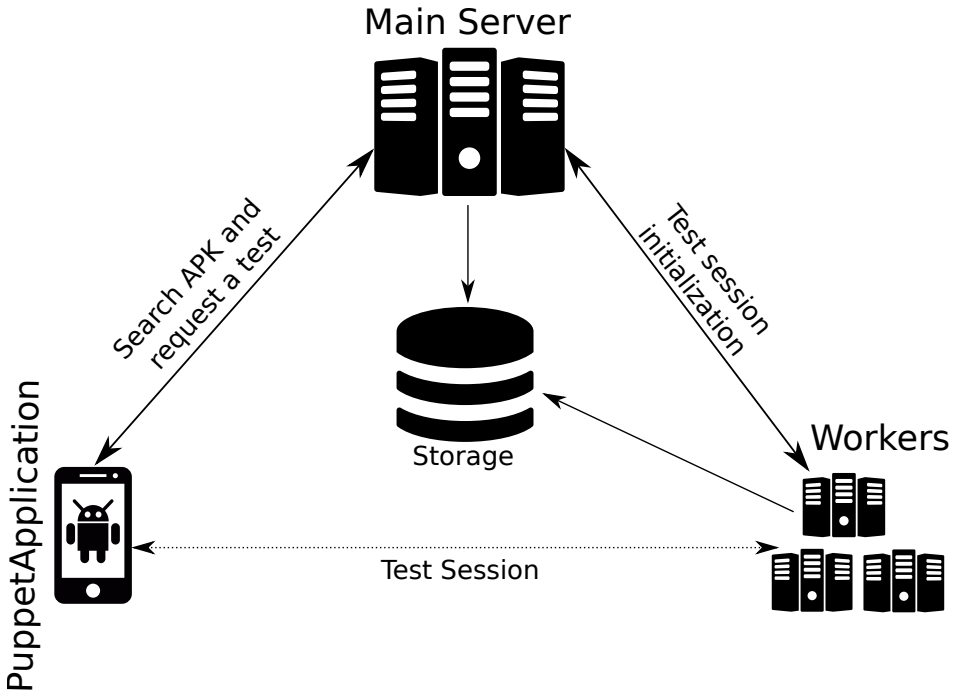


FIGURE 3.6: The PuppetApplication sends requests to the Main Server which forwards the job to the Worker(s). After the initialization, the PuppetApplication and a Worker estabilish a VNC session.

The data sent to the server are the events injected by the users, encoded in an apposite way, along with the coordinates where the event took place or, in the swipe case, a sampling of all the coordinates involved (namely one pair of numbers each eight). Moreover, the coordinates are pre-processed before packing them into a message, since a rescaling is needed due to the (possibly) different size between the client emulator screens.
The data received from the server are the images taken from the emulator framebuffer, rescaled to match the device resolution, such that the user "feels" the tested application as if it were running locally.

The VNC client is embedded in the PuppetApplication and its code mainly comes from "tvnjviewer," a Java porting of TightVNC [17]. We slightly modified the code to allow a full-screen visualization of the `PuppetdroidViewerActivity`, namely the `Activity` where the images coming from the VNC server are drawn.

The PuppetApplication is written in Java leveraging the Android SDK [18].

3.3.2 Worker

The *Worker* is responsible of the three main phases described in *Section* 3.1.

Phase 2 (Recording) As a test session is requested, the Main Server sends to the Worker the information about the application to exercise. The *Worker* can then initialise the virtual environment to host the test session: the application is installed and started. At this point the recording routine is started and the user is prompted with the application to test wrapped in the PuppetApplication. While the user interacts with the application, the recorder (see *Section* reftec:rec creates the graph, interaction after interaction. When the test session is over, the resulting graph is stored by the Worker.

Phase 3 (UI-Similarity graph lookup) When a rerun is requested, the target application is provided to the Worker. Like in the recording phase the virtual environment is initialised and the application is installed and launched on the emulator. At this point the lookup phase takes place: as the application is started, the Worker takes a screenshot of the current view of the application, selects the best cluster by means of the hamming distance between the screenshot and the centroids pHashes, and randomly selects a graph from the prerecorded ones belonging to the applications that are within the cluster.

Phase 4 (Re-execution) After the graph is selected, the re-runner tool is called and the graph is re-executed on the target application. After a settable timeout the re-execution is considered over. At is point the results of the rerun, namely a binary file generated by CopperDroid, are stored for future analysis.

3.4 Technical details

DriveDroid is a distributed system whose components run on different machines. In this section we describe the technical details of the system.

3.4.1 Main Server

The Main Server is responsible for managing the test requests from the clients and their enqueuement in a collection in the database used as a persistent concurrent queue (see *Section* 3.4.7 and *Section* 3.4.8). It also continuously monitors the queue state to be able to acknowledge when a Worker pulls a job (which means the worker is available for that

particular stimulation session), in order to forward the client info to that Worker, and to forward the Worker info to the client, making possible a VNC connection.
The whole subsystem supports multiple connections from both clients and Workers thanks to the multithreading and the competitive queue.

The Main Server is written in Python 2, leveraging the `multiprocessing` library for managing multiple connections.

3.4.2 Worker

The Worker is the component where the target applications are actually executed in Android virtual devices (AVDs). It is also responsible for pulling the user-requested jobs from a concurrent queue, dispatching them to the available AVDs and for saving the results.

3.4.2.1 AVD

The AVD is a software emulator capable of running an Android image; it is provided directly by Google within the Android SDK. In our case we used a slightly modified AVD based on CopperDroid, a dynamic-analysis system that extracts OS-level events such as system calls and reconstructs their Android-level semantics and data. At the end of each execution a CopperDroid surrogate provides a list of the detected behaviours (e.g., place a phone call, access to the network interfaces, etc.) by means of system calls sequences. A pool of AVDs is started at the Worker boot and after a test session an emulator is always killed, recreated and restarted from cold to ensure it is perfectly clean for the next test.

3.4.2.2 VNC Server

Within the AVD the target application is running during **Phase 2 (Recording)**, waiting for user inputs. The component responsible for receiving input events from the PuppetApplication is the VNC Server. It is a native coded application (it need to be as responsive as possible) installed and started as a daemon just after the target application to receive and inject into the running target application the events coming from the client and taking images from the framebuffer at high rates in order to send them back to the client: this way on the mobile phone the user has the impression of a continuous update of the application as if it were running locally. The way the events are injected is by leveraging UI-Automator [19].

3.4.3 Recorder

The recorder implements the core of **Phase 2 (Recording)**. It is the component that intercepts the user inputs before they reach the VNC server in order to dump a compact representation of them. The output is a graph with states and interactions, as described in *Section* 3.2.2. The XML tree(s) of the application views are inspected in order to bind each state of the graph to a specific view of the APK; a separated thread looks for pop-ups like `AlertDialogs` in order to take them in to account as well in the views hierarchy.

3.4.4 Rerunner

The rerunner implements **Phase 3 (UI-similarity graph lookup)** and **Phase 4 (Re-execution)**. It is responsible for fetching a recorded graph that contains the representation of the events to send to UI-Automator for the injection in the running sample. It is also the responsible for checking whether a stuck state has been reached and a backtrack is needed in order to choose another path to explore with new events to inject.

The Worker is written in Python 2, leveraging the `multiprocessing` library for managing multiple connections.

3.4.5 Web application

The web application serves as a simple administrative interface for the system. It allows users to upload new applications and to inspect the APKs dataset and the recorded graphs. The web application leverages *aapt* [20] to determine whether the uploaded applications are well-formed APKs and to extract the icons and a copy of their manifest. Well-formed applications are identified with the SHA-256 of the APK file and stored in the persistent data tier (see *Section* 3.4.7).

3.4.6 Public key infrastructure

In order to ensure data confidentiality and integrity and to provide authentication both from server and client side in all the communications, we employed a public key infrastructure (PKI). The hierarchical structure is shown in *Figure* 3.7. The PKI leverages OpenSSL [21] and all the configuration details/certificate constraints and policies are

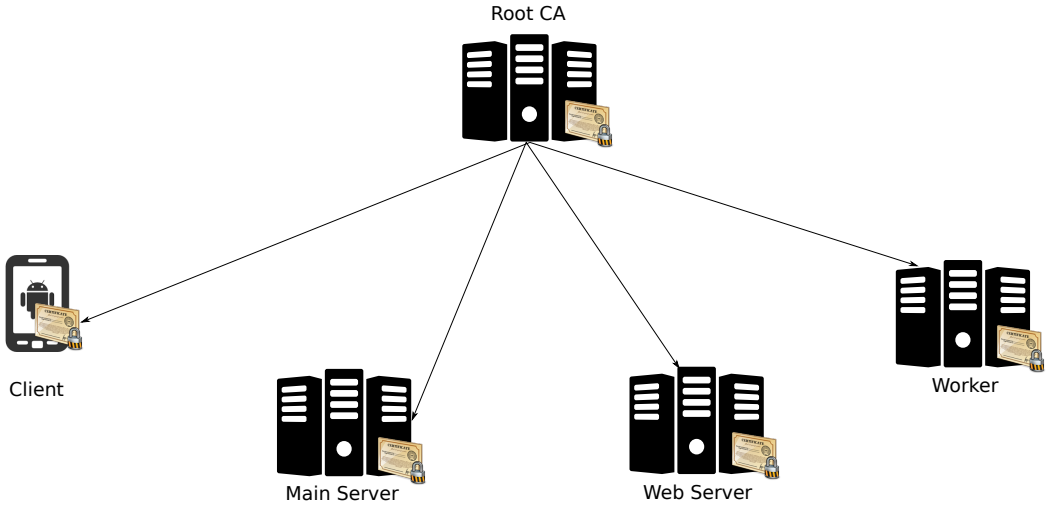


FIGURE 3.7: DRIVEDROID PKI: at the top the root CA, at the bottom the client, the servers and the workers.

stored in the *openssl.cnf* file. All the keypairs are generated with *elliptic curves cryptography* leveraging the *secp521r1* elliptic curve [22] for performance and security reasons.

3.4.7 Persistent data management

The persistent data needed by DRIVEDROIDare stored either in a database or in a file server, depending on their nature, for ease of retrieval.

Database

The database stores all the metadata. It is based on MongoDB, a document-oriented database engine. The documents' scheme is summarised in *Figure* 3.8.

File Server

The file server, implemented by a file system structure, stores the APKs, their icons and manifests, the recorded graphs (in the form of pickle files) and the CopperDroid generated files.

3.4.8 Use cases

Use case 1: Registration

The first time the PuppetApplication is installed on a smartphone the user has to go through the registration process:

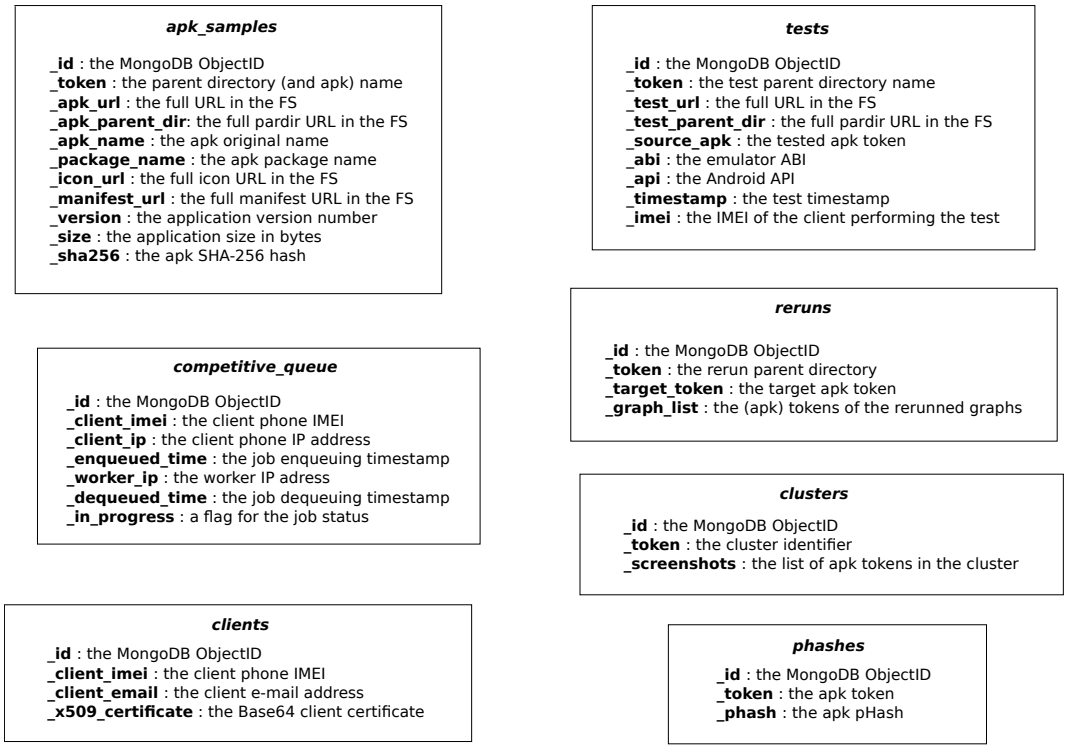


FIGURE 3.8: The database collections schemas.

1. The user inserts all the data required for the CA to issue a certificate: Country code, State or Province name, City name, Organization name, Organization Unit name, Common name as well as a password (which must be confirmed) to protect the soon generated keystore. The Common Name must be a syntactically well-formed e-mail address, while the password must be at least 8 characters long with at least a letter and a number.

2. The application generates a keystore and a keypair; all the material is stored into the former and the private key from the latter as well as the user inserted data are employed to generate a Certificate Signing Request.

3. The CSR is sent via a server only authenticated socket session (the cacert is embedded into the application for certificate pinning purposes) to the CA Server.

4. The CA Server for security reasons overrides the X509v3 extensions with the ones specified for the clients in the configuration, and signs the certificate.

5. The CA Server stores into the MongoDB Database (see *Section* 3.4.7) the client IMEI, the client e-mail and a reference to the certificate, while the certificate itself is saved in a dedicated archive.

6. The certificate is sent back to the client which responds with an ACK successfully terminating the process.

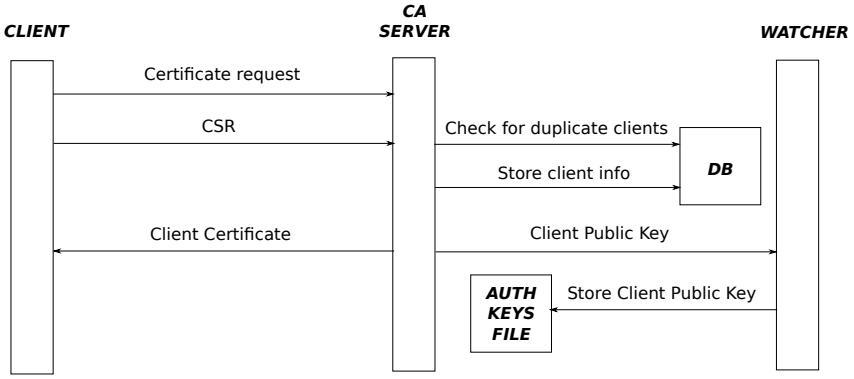


FIGURE 3.9: The client generates and sends a CSR to the CA which signs it, generates a client certificate and forwards the client public key to the Main Server Watcher.

7. The CA Server extracts the client public key from the client CSR in order to send it to a Watcher Server running along with the Main Server.

8. The Watcher takes the key, converts it in the OpenSSH format and appends it on a new line in the Main Server ~/.ssh/authorized_keys file, since every communication except the client registration and the Webapp is under SSH tunnelling.

Use case 2: APK search

From the main view of the PuppetApplication the user can perform a search between all the uploaded APKs:

1. The user types into the apposite EditText the keyword he wants to search for and confirms the search.

2. The application checks for an already opened SSH Tunnel with the Main Server: if not found or not active a new one is opened. After that a TLS socket is opened on TCP port 5901 with a first message ("com.necst.androvnc").

3. The GET_APK_LIST value and the search string are forwarded to the Main Server.

4. The Main Server receives the search keyword and looks for one or more regex matches the package names and the application names among all the samples metadata in the Database.

5. The Main Server, for each results, sends back all the metadata retrieved from the Database, except for the data regarding the tests already done; in case no results were found, a special value is returned instead.

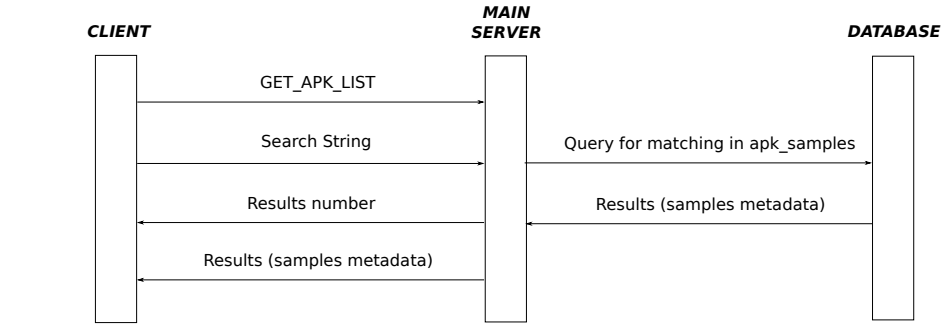


FIGURE 3.10: The client sends a search string to the Main Server which queries the database and returns the matching results.

6. The client receives all the data and presents them to the user with a vertical list with every entry clickable in order to pop-up an `AlertDialog` with more detailed information and a button to start a test on the selected sample.

Use case 3: Test session

From the `AlertDialog` pop-up explained above the user can request a test session:

1. The user confirms the test request.
2. The application checks for an already opened SSH Tunnel with the Main Server: if not found or not active a new one is opened. After that a TLS socket is opened on TCP port 5901 with a first message ("com.necst.androvnc").
3. The CONNECT value is sent to the Main Server along with the token of the sample to test.
4. The Main Server enqueues the job in the competitive queue and waits for a Worker to pull it. If after a minute nothing has happened the job is automatically dequeued and the procedure aborted; the client is informed that the system has no sufficient resources to execute the test and the session is over.
5. If a Worker pulls the job, it writes into the competitive queue its IP address; as soon as the Main Server notices this, it fetches the Worker IP, plus deleting the queue entry.
6. The Main Server fetches the sample metadata from the Database and the client public key from the authorized keys file; it also creates a new entry in the tests collection; then it opens a TLS socket with the Worker on port 6903.

7. The Main Server sends to the Worker the application metadata and the client public key.

8. The Worker adds the public key in its authorized keys file, creates the test folder in the File Server, fetches the sample thanks to the metadata, installs it on the first available AVD and sends to the Main Server the TCP port that will be used server side for the VNC session. Meanwhile it also attaches the recorder and CopperDroid to the running application process.

9. The Main Server forwards the Worker IP and port to the client, then closes both the connections with the client and the Worker becoming from that moment agnostic about everything happens between the other two parts.

10. The client application checks for an already opened SSH Tunnel with the Worker: if not found or not active a new one is opened. After that a TLS socket is opened and the VNC session starts.

11. The VNC session proceeds until the user chooses to disconnect by clicking the apposite button: a DISCONNECT value is sent to the Worker and the client VNC closes the session.

12. In the Worker the recorder finalizes and stores the recorded graph in the File Server, then stops and detaches itself from the running sample process; the system also stops CopperDroid and dumps the whole recorded stream to a binary ".copper" file in the test folder.

13. The AVD is restarted and the Play Services installed; the client public key is removed from the authorized keys file.

In case an error happens client side, an ABORT value is sent to the Worker (and to the Main Server if still listening). This causes the complete rollback of any changes made after the very first client request. The same happens if the error triggers on the Main Server side, while if the problem is within the Worker, the Worker itself sends to the client (via Main Server if before the VNC session) an appropriate error code. Finally if the VNC session remains inactive for more than one minute, the Worker declares the test aborted and closes the connection.

Use case 4: Re-execution

The re-execution consists in stimulating an application by generating inputs according to a previously recorded, perhaps on a different sample, graph. It requires a simple client socket that communicates with the Worker in order to tell him what rerun on which APK. Its workflow is the following:

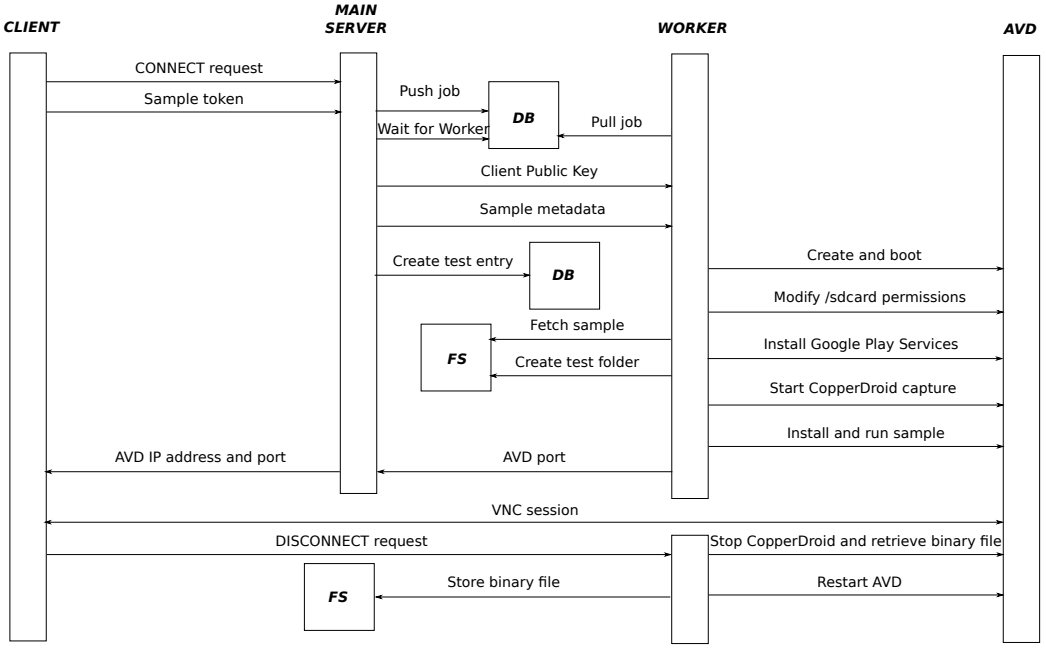


FIGURE 3.11: The client sends a test request to the Main Server which forwards the job to an available worker. After the initialization, the VNC session takes place. When the client disconnects, the worker halts the test and stores the results.

1. The user chooses to do the re-execution from the socket client, specifying the target application.

2. The worker receives the request and, if not already present, starts a CopperDroid instrumented emulator.

3. The target application is installed, a screenshot is taken and classified (leveraging the pHash and the hamming distance metric) into a cluster.

4. The Worker then retrieves from the database the metadata of the cluster the screenshot has been classified in, and randomly selects all the graphs belonging to one of the samples present in that cluster.

5. The Worker merges together the graphs into a single one, aiming for coverage maximization.

6. The Rerunner component of the Worker starts parsing the graph and sending injectable events to UI-Automator which passes them to the running target application.

7. After 7 minutes the Rerunner and CopperDroid stop.

8. A binary ".copper" file is created and stored, and the emulator is restarted with the same procedure adopted for the normal tests.

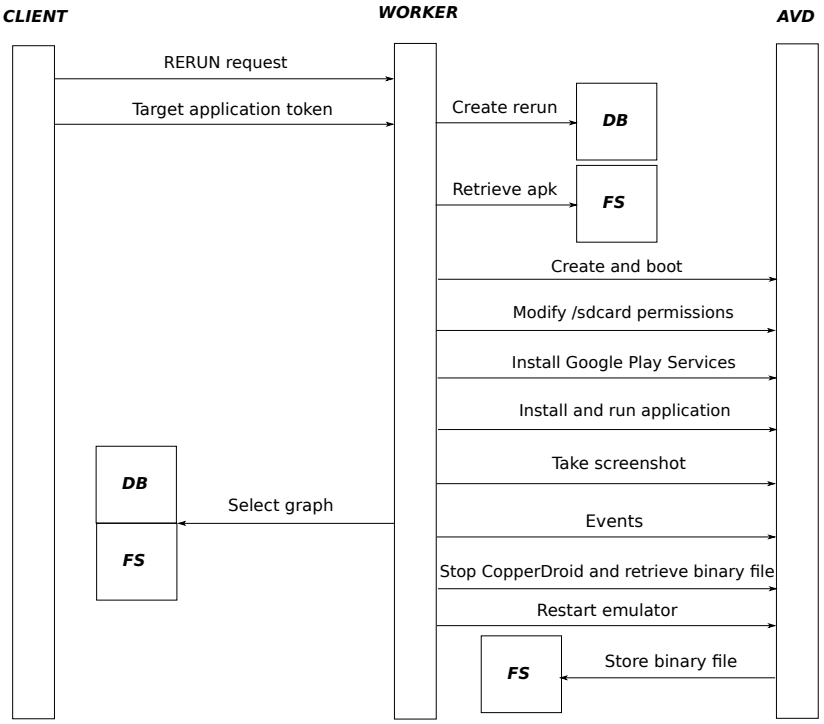


FIGURE 3.12: The client sends a re-execution request to the Worker which retrieves a graph and uses it as a guide to stimulate the target application. After 7 minutes the Worker halts the re-execution and stores the results.

Chapter 4

Experimental evaluation

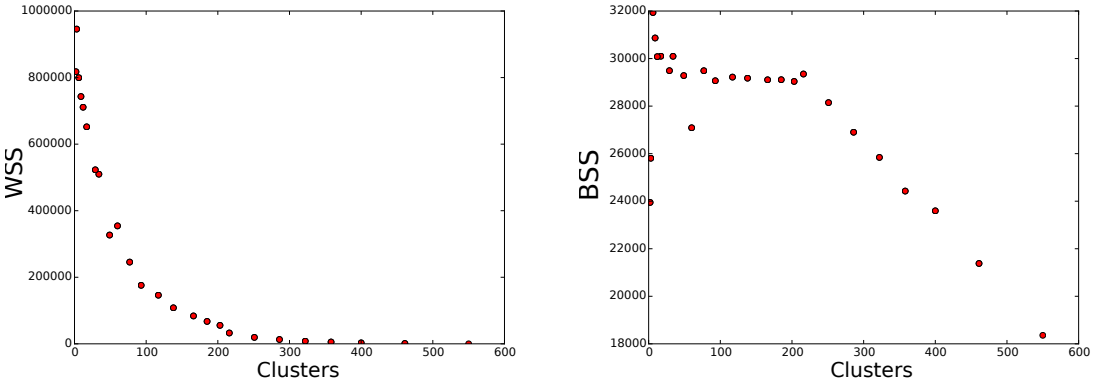
In this chapter we present the experiments we performed in order to evaluate our approach. In *Section* 3.1, we present an evaluation of both the Human and Markovian approaches against Monkey (*Section* 4.3.2 and *Section* 4.3.3). In *Section* 4.4), we provide a proof of concept of how our approach can deal with extreme user-interaction-based evasion techniques showing how Monkey fails in this case. Finally, in *Section* 4.2), we provide a performance evaluation both for the clustering algorithm and the **Phase 3 (UI-similarity graph lookup)**.

4.1 Dataset

We obtained applications for our dataset from the Google Play Store [23] and third party markets (Apptown [24] and 1mobile [25]). We also had access to the Android Malware Genome Project [26]. Overall, our dataset comprises 1,800, of which 100 from the Google Play Store, 200 from Apptown, 150 from 1mobile, and 1,350 from the Android Malware Genome Project.

For each application in our dataset we took the screenshot of the first view as required by **Phase 1 (Clustering)** and **Phase 3 (UI-similarity graph lookup)**, which are both based on the first view to identify an application. Many applications could not start or just crashed, so we removed them from the dataset, deeming them useless since not testable: we collected 1,028 screenshot associated to 1,028 corresponding applications.

We ran **Phase 1 (Clustering)** to cluster our dataset. As shown in *Figure* 4.1, we tuned the clustering using the knee-elbow method (see *Paragraph* 3.2.1), using both the within and between sum of square. According to the knee-elbow results, we tuned our clustering obtaining 206 clusters.



(A) Within sum of squares (B) Between sum of squares

FIGURE 4.1: Knee-elbow to tune and evaluate clustering with within and between sum of square metrics.

4.2 Experiment 1: Clustering and lookup evaluation (Phase 1 and 3)

In this section we provide an evaluation of the clustering algorithm defined in (*Section* 4.2.2) and of the lookup approach described in (*Section* 4.2.3). In particular, for the latter, we evaluated two different techniques, Naïve and Centroid, comparing them.

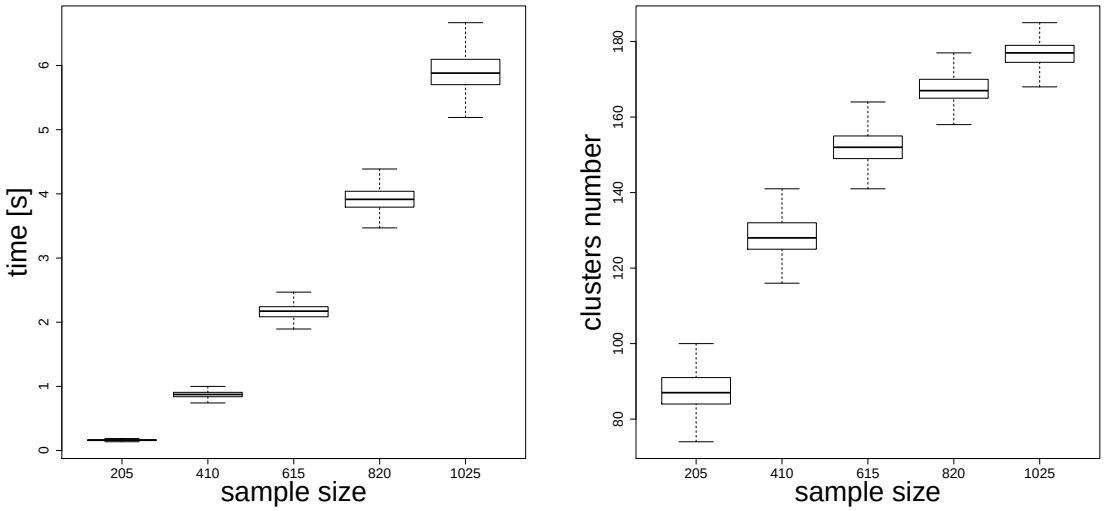
4.2.1 Experimental set-up

For all the experiments in this section, we used a machine with 2 cores and 4GB of RAM. We partitioned our dataset in 5 partitions of different size: 205, 410, 615, 820 and 1,025 screenshots.

4.2.2 Clustering evaluation

In this section, we evaluate our clustering performance drawing 500 random samples and, for each random sample, measuring the time required by the clustering algorithm to complete.

In *Figure* 4.2 we provide a box plot of the results. The results confirm the complexity of our algorithm is polynomial in the number of elements, in particular it grows as n^3 where n is the number of applications in the dataset. Indeed, looking at the median of each boxplot, it is evident how the growth of time follows the n^3 polynomial function. We could confirm that the theoretical complexity of the hierarchical clustering that is exactly $O(n^3)$; in principle it is possible to lower it to $O(n^2 logn)$ implementing a priority



(A) clustering time variation (B) clusters number variation

FIGURE 4.2: Clustering Evaluation: boxplot showing how the clustering time and the number of clusters vary in function of the dataset size

queue, but we could not due to how our problem is defined: in a non-metric system like the pHashes-with-hamming-distance one, is hard to define an ordering, which is the base on the top of which a priority queue is built.

Finally, in *Figure* 4.2b we evaluated how the number of clusters varies according to different size of the dataset. From the results we can see that each cluster contains on average 4 applications. This means that with just one graph recorded, we are able to re-execute 4 applications on average. When clustering 1,028 applications, we obtained 216 clusters with about 5 applications on average.

4.2.3 Lookup performance evaluation

In this section we compare the performance of the two lookup techniques (naïve and centroid based). As done for *Section* 4.2.2, we draw 500 random samples and for each random sample we measure and compare the time required by both the approaches. We define the *distance-calculation time* as the time taken by the lookup procedure to compute all the distances between the clusters and the query point. Note that in the case of the naïve lookup strategy, this is the time required to calculate the distance between the query point and every point in the dataset. We define the *distance-minimization time* as the time taken by the lookup procedure to choose the nearest cluster. We define the *total time* as the sum of the distance-calculation and the distance-minimization times.

In *Figure* 4.3, we show the evaluation of both the naïve and centroid based techniques. It is clear how the centroid based performs better than the naïve: comparing the total time of the two techniques (*Figure* 4.3a and 4.3d), we can say that the centroid based technique is on average the 79% faster than the naïve. The reason of such difference, is due to the way the distance from the clusters is computed (i.e, distance-calculation time). We recall that, differently from the naïve technique, that spans across the entire dataset, the centroid based requires number of comparisons equal to the number of clusters only.

As mentioned before, we find the performance difference between the two techniques in the distance-calculation time. As shown in *Figure* 4.3b and *Figure* 4.3e the naïve distance-calculation time is in the order of hundreds, while the centroid based one is in the order of tens. However, for both the techniques, the growth of the distance-calculation time is linear, in the number of samples for the naïve and in the number of clusters for the centroid-based.

In *Figure* 4.3c and *Figure* 4.3f, we can see how the distance-minimization time is comparable for both the techniques. The tendency is linear in the number of clusters, since once the distance-calculation time is computed, we have to find a minimum over a set of distances: one for each cluster.

4.3 Experiment 2: Recording and re-execution evaluation (Phase 2 and 4)

With this experimental evaluation we want to validate our approach, comparing how many behaviours it triggers with respect to Monkey.

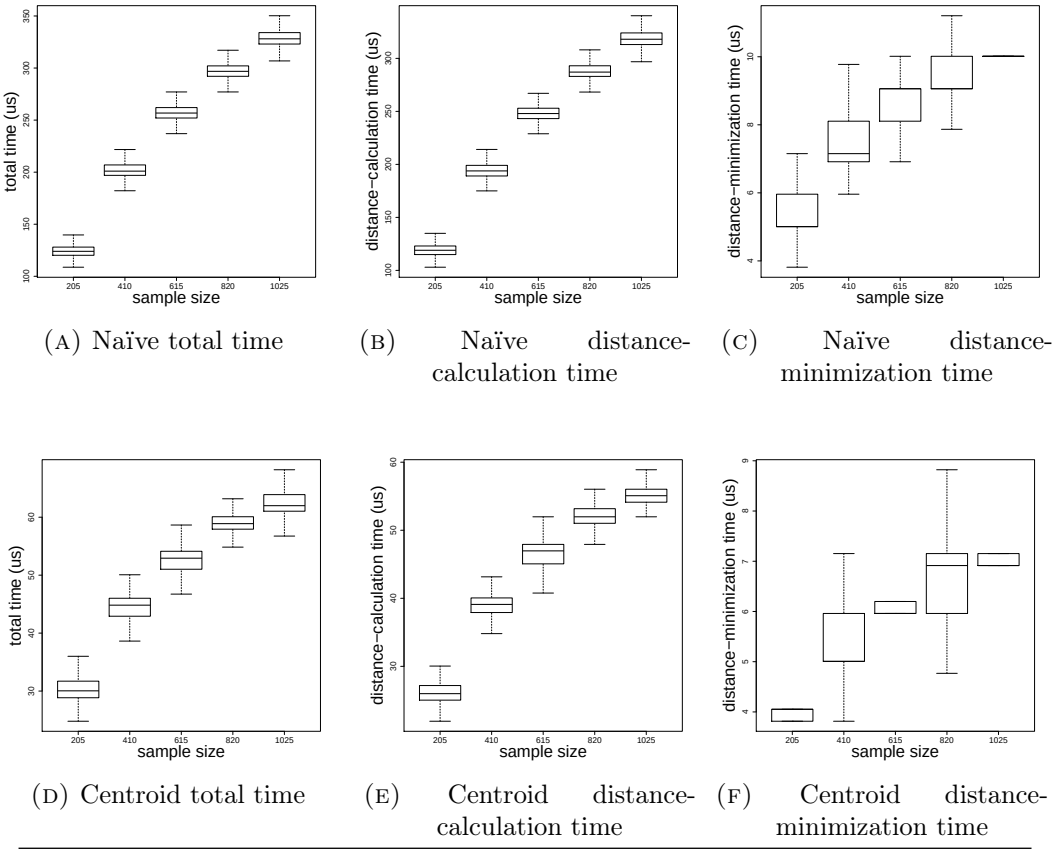
First, we evaluate whether a pure human stimulation is better than Monkey, second we evaluate the actual approach, rerunning the recorded interaction over the whole dataset and comparing it again with Monkey.

In particular, the first phase is explained in *Section* 4.3.2, while in *Section* 4.3.3 we present the second phase.

4.3.1 Experimental set-up

We leveraged the last version of *CopperDroid* [14] to extract behaviours at runtime, while the system name drives the execution.

We performed the following tests:



(A) Naïve total time (B) Naïve distance-calculation time (C) Naïve distance-minimization time

(D) Centroid total time (E) Centroid distance-calculation time (F) Centroid distance-minimization time

FIGURE 4.3: Box plot showing how the different execution times vary according to the different size of the dataset, both for the naïve and centroid techniques.

- We stimulated 1,028 applications with the Monkey approach, injecting 5,000 events.
- We started **Phase 2 (Recording)** (see *Section* 3.1.2 and *Paragraph* 3.2.2), letting users stimulate 216 applications (once for each cluster).
- We started **Phase 4 (Re-execution)** (see *Section* 3.1.4 and *Paragraph* 3.2.2), use 1,028 different applications as targets.

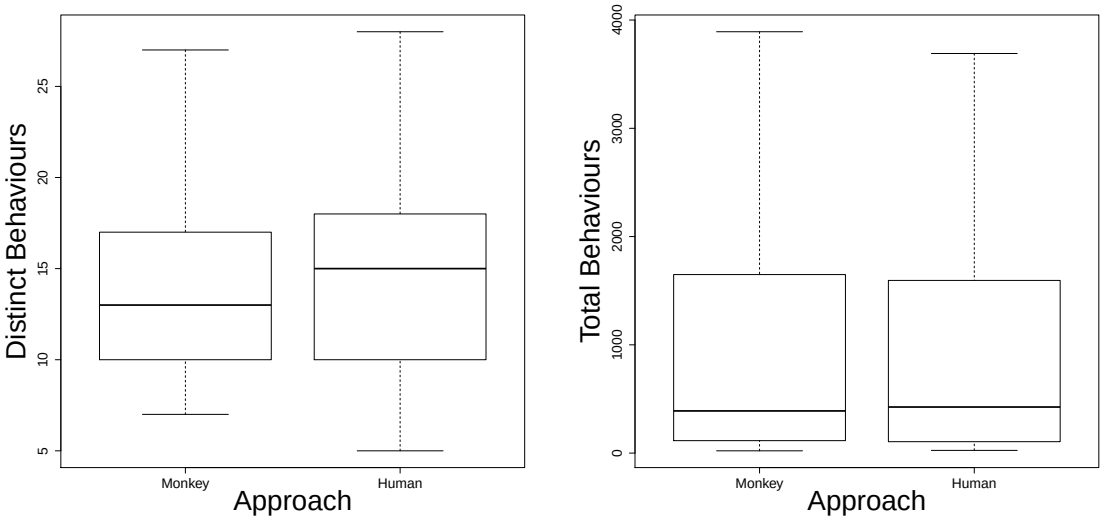
  1,028 Markovian using the 216 graph recorded.

However, during the tests, we found 13 applications not working properly, thus we reduced the dataset to 1,015 applications, of which 203 having a valid stimulation performed by a human user.

For the Monkey approach we set-up a timeout of 7 minutes, since Monkey sometimes hangs. The stimulation performed by the users varied from 5 to 7 minutes, according to the application complexity. For the graph re-execution, we set-up a timeout of 7 minutes, in order to have the same time given to Monkey.

ID	Description	Age	Average stimulation time	Apps
Person 1	Android Security student	24	6 minutes	58
Person 2	Android Security student	24	6 minutes	58
Person 3	System Security student	24	6 minutes	20
Person 4	Android Security student	24	6 minutes	20
Person 5	Common user	18	6 minutes	20
Person 6	Common user	55	6 minutes	20
Person 7	Common user	56	6 minutes	20

TABLE 4.1: Testers involved in the graphs recording process.



(A) Number of distinct behaviours (B) Number of total behaviours

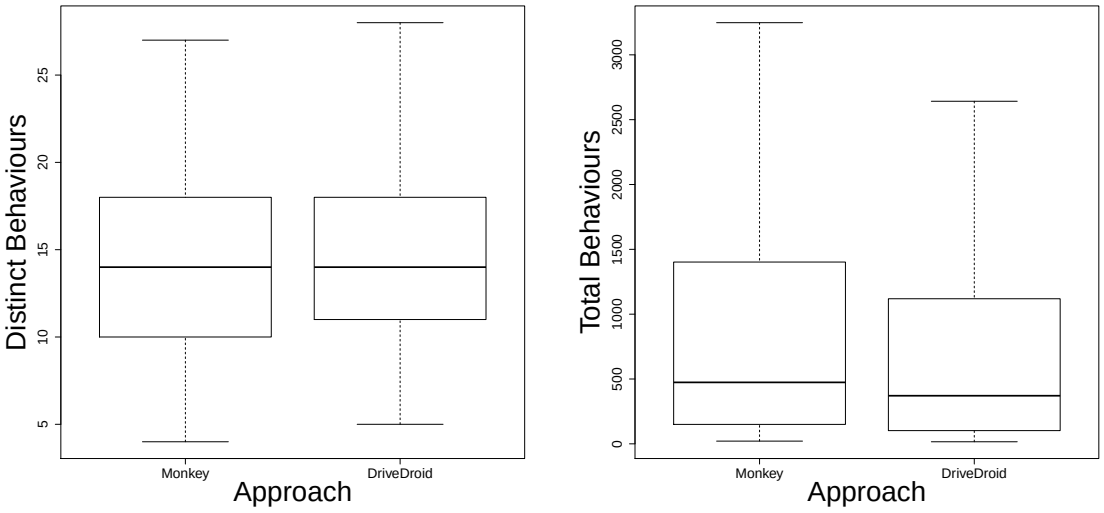
FIGURE 4.4: Comparing the number of behaviours triggered by the Human and Monkey approaches.

To host the emulator session, we used a machine with 16 cores, 30 Gigabytes of RAM and two 160 Gigabytes SSD. We selected a pool of users, including ourselves, to stimulate the applications (see *Table* 4.1). We instructed the users to stimulate the application striving to maximize the functionalities explored. All the people mentioned performed the tests on Google Nexus 5 phones.

In the next two sections, we present the results obtained by comparing Monkey with two approaches: Human (*Section* 4.3.2) and DRIVEDROID (*Section* 4.3.3).

4.3.2 Human vs. Monkey

We present here the results of our evaluation. In *Figure* 4.4a and in *Figure* 4.4b we provide a boxplot of the results.



(A) Number of distinct behaviours (B) Number of total behaviours

FIGURE 4.5: Comparing the number of behaviours triggered by DRIVEDROID and Monkey approaches.

As expected, the results showed that human approach can perform, in terms of stimulating distinct behaviours, at least as good as Monkey and that, on average, it stimulates more distinct behaviours.

However, we noticed the two approaches are very close comparing the total behaviours. This is due the fact that injecting 5,000 events, Monkey can stimulate over and over the same paths, but it fails when it comes to stimulate different ones. A human instead, in the same time of Monkey, cannot inject all these events, but they can lead to a greater code coverage (more distinct behaviours), which is the goal we wanted to achieve.

4.3.3 Re-execution vs. Monkey

We present here the comparison of our re-execution approach versus a random one, implemented through Monkey. In *Figure* 4.5a and in *Figure* 4.5b, we provide boxplots of the results.

We found that DRIVEDROID can trigger at least as many behaviours as Monkey, slightly better. Again, the standard deviation confirms that there is not a high dispersion in the data.

In this case Monkey can trigger more total behaviours than DRIVEDROID. However, this is easily explained by the fact that the DRIVEDROID is faithful to a human stimulation, thus resulting in few total events injected compared to Monkey. However, trigger the

same behaviours more than once is not our aim: indeed our goal is to trigger as many distinct behaviours as possible.

4.4 Experiment 3: Interactive challenge evaluation

In *Section* 2.2.2 we defined the concept of interactive challenge and we described an application implementing such evasion technique. In this section we describe the evaluation we did to show that random stimulation approaches cannot deal with this kind of applications.

4.4.1 Dataset

The dataset is a set of 6 APKs of the type explained in *Section* 2.2.2. We recall that these applications consist in a user-interaction-based challenge: in order to trigger the true behaviours, a user has to execute a particular sequence of actions on the screen. Five of these applications are clustered together, while the last one is used as source for the rerun.

4.4.2 Experimental set-up

We collected a stimulation graph from one of the samples and we re-executed it on the other ones while recording the behaviours with a CopperDroid instrumented emulator. We also performed Monkey stimulation on the same applications; we used the same procedure described in *Section* 4.3.3.

4.4.3 Results

Both the human stimulated sample and all the re-executions triggered 16 different behaviours, while Monkey in every run could not trigger any behaviour due to not passing the interactive challenge. We performed the experiment on an AVD with the graphical interface, so we could identify when and where Monkey fails: from our observations Monkey was never able to complete the second step (the second tap), which is expected, since the probability is $(1/24)^2$ which is 1/576, around 0.17%. In all but one cases even the first tap was out of the correct place, and even that is expected, being the probability of centering the right corner around 4.2%.

Chapter 5

Limitations and future work

In this chapter we analyse the main limitations of DriveDroid. In *Section* 5.1, we explain the main limitations of our approach. In *Section* 5.2, we discuss the limitation of the current prototype. In *Section* 5.3 we describe the possible future extensions of this work.

5.1 Approach limitations

In this section we discuss the main limitations of our approach: facing login barriers, dealing with a weak user stimulation and the problem of looping over the same paths while re-executing.

Registration and login barriers During **Phase 4 (Re-execution)**, it is possible that an application shows a login form. If it is the case, it is not possible for the system to go through the login barrier since we cannot guess the credentials to login. It results in not exploring entire views and code path that would be reached after the login. Facing this problem is far from trivial, most of the solutions are ad-hoc ones, for example it would be possible to register an account for the system and use its credentials for a login. Another alternative could be modify the application code in order to by-pass the wall. However all these solutions are strongly dependent on the context.

Poor human stimulations If the graph recorded during **Phase 2 (Recording)** is incomplete, for instance due to a test too short in time or a lazy user, it will likely not stimulate many behaviours during **Phase 4 (Re-execution**. Due to the crowd-sourcing approach this is an actual issue. However, we partially mitigate this limitation by merging all the graphs recorded for an applications.

Loops Due to the probabilistic elements in the approach, it is possible that the re-execution keeps looping between the same states, not exploring new paths. A possibility to mitigate this problem is to identify when a loop occurs and force the re-execution to explore other paths. We are currently working on it and it is a step forward for extending the current version of the approach.

5.2 Implementation limitations

In this section, we explain some limitation we found in our implementation.

Emulated environment detection
An increasing number of malware is able to detect a sandboxed running environment and, thus, to refuse running or to hide some (malicious) behaviours [27]. Since DroidDrive uses an emulator for running an application, it is possible that some malicious behaviours are not shown since the malware detects the virtual environment. There is no solution to this problem. It is a race between malware analysts and authors: as a solution to hide the emulator environment is found, in turn a solution to evade it is found as well. Emulator evasion is still an open problem for the research field of dynamic malware analysis. In this work we focus on an higher-level, and perhaps harder, form of evasion, that is the detection of non-human users, which will persist even on non-emulated environments, and thus needs more attention from the research community.

Images similarity
To implement our similarity metric we leveraged perceptual hashing: for our needs it was one of the best options, since it is very fast and one of the simplest approaches we could find in literature. However, we identified some practical limitation due to the fact that, despite the pHash is robust to image variation, sometimes it is too sensible, resulting in imprecise calculation of the distance between two screenshot that should be considered similar. A future implementation of our approach could leverage other techniques.

Applications similarity
In our implementation we assumed two sample similar if their first view is similar. This choice is a limitation since it is possible that two applications have different initial views and similar internal views (false negative) or that they have similar initial view and different internal views (true positive). As matter of fact we found few different instances of the same malware all with similar internal views but with a completely different first

view. To overcome this limitation, a solution could be switching the graph at every state transition, changing the screenshot granularity to one for each view.

5.3 Future work

One of the main limitations of our approach is the possibility of the re-execution remaining stuck on loops, due to the probabilistic selection of the edge to follow from a graph node. To this end, future research directions include modifying the re-execution algorithm to detect and react to loops. For example, a re-execution that keeps exploring the same path(s) could be forcefully restarted (with or without a new graph) or instructed to not taking that/those path(s) again.

On the similarity side, a possible extension to our work consists in considering more screens for the distance computation. It is also possible to guide this process by means of static analysis: parsing the layout files may be useful to create an a-priori model to support the decision on when taking a new screenshot, without overfitting (i.e., taking a screenshot when insignificant UI modifications happen). Furthermore, this can be used to state in a fast way that two applications are not similar (large imbalance in the amount of layout files present in the applications and/or significant differences in their structure).

Finally, on the implementation side, an important improvement would be working on the VNC (both client and server), ensuring compatibility with the newest Android versions and providing a better user experience. In particular, to fully leverage the crowd-sourcing possibilities of DriveDroid, it is necessary to cut down the VNC latency, for example by sending only a "diff" file of each image from the server to the client side.

Chapter 6

Conclusions

The goal of our work was to contribute to the solution of the problem of stimulating Android applications. We proposed a model to represent the whole process a user performs to exercise an application and represented the it as a directed graph where the node are the views of the applications and the edges are the actions an user can perform from that view. We designed a novel strategy consisting in generating a graph from the UI interaction a user provided, and reproducing those interactions over other similar applications.

We evaluated our approach and we conducted experiments on our dataset. In particular, we showed that DriveDroid can stimulate at least as many behaviours as the main state-of-the-art approaches. Moreover, we showed how our approach can deal with some extreme evasion techniques based on an heavy user interaction. Results showed that DriveDroid can reuse traces recorded by users with a factor of 5 in average, thus multiplying the effort of the human analyst: we strongly believe that creating a crowd-based, user-centric dynamic malware-analysis system is very promising and feasible as an extension of our work.

Despite the encouraging results, there are open points that we hope will be subject of future work. The main one consist in forms which require the knowledge of some secret data to proceed; the only possible way to deal with this problem (except for backtracking to the previous view) is temporary stopping the automatic stimulation and ask for the help of a human user. Another problem are the loops during the re-execution, which can be countered with loop-detection techniques. Finally, an issue arises when a graph is recorded lazily by an user: we could only mitigate this problem by merging different graph recorded by different users on the same application.

Bibliography

[1] International Data Corporation (IDC). Smartphone OS Market Share, 2015 Q2. URL https://www.idc.com/prodserv/smartphone-os-market-share.jsp.

[2] H. Lockheimer. Android and Security, 2012. URL http://googlemobile.blogspot.co.uk/2012/02/android-and-security.html.

[3] 360 Mobile Security LTD. Q2 2015 Android Malware and Vulnerability Report. Technical report, 360 Mobile Security, July 2015. URL http://www.slideshare.net/360Security/q2-2015-android-malware-and-vulnerability-report-360-security.

[4] Google Inc. UI/Application Exerciser Monkey. URL http://developer.android.com/tools/help/monkey.html.

[5] Andrea Gianazza, Federico Maggi, Aristide Fattori, Lorenzo Cavallaro, and Stefano Zanero. PuppetDroid: A User-Centric UI Exerciser for Automatic Dynamic Analysis of Similar Android Applications. *CoRR*, abs/1402.4826, 2014. URL http://arxiv.org/abs/1402.4826.

[6] Google Inc. Android Dashboard. URL https://developer.android.com/about/dashboards/index.html.

[7] Google Inc. Android UI, . URL http://developer.android.com/guide/topics/ui/overview.html.

[8] Google Inc. Automatic User Interface Tests, . URL http://developer.android.com/training/testing/ui-testing/index.html.

[9] GDATA. Mobile Malware Report. URL https://public.gdatasoftware.com/Presse/Publikationen/Malware_Reports/G_DATA_MobileMWR_Q1_2015_US.pdf.

[10] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. ANDRUBIS - 1,000,000 Apps Later: A View on Current Android Malware Behaviors. September 2014. URL https://iseclab.org/papers/andrubis_badgers14.pdf.

[11] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: an input generation system for android apps. In Bertrand Meyer, Luciano Baresi, and Mira Mezini, editors, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 224–234. ACM, 2013. ISBN 978-1-4503-2237-9. doi: 10.1145/2491411.2491450. URL `http://doi.acm.org/10.1145/2491411.2491450`.

[12] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In Ting Yu, William Enck, and Xuxian Jiang, editors, *SPSM'12, Proceedings of the Workshop on Security and Privacy in Smartphones and Mobile Devices, Co-located with CCS 2012, October 19, 2012, Raleigh, NC, USA*, pages 93–104. ACM, 2012. ISBN 978-1-4503-1666-8. doi: 10.1145/2381934.2381950. URL `http://doi.acm.org/10.1145/2381934.2381950`.

[13] Michael Bierma, Eric Gustafson, Jeremy Erickson, David Fritz, and Yung Ryn Choe. Andlantis: Large-scale android dynamic analysis. *CoRR*, abs/1410.7751, 2014. URL `http://arxiv.org/abs/1410.7751`.

[14] Kimberly Tam, Salahuddin J. Khan, Aristide Fattori, and Lorenzo Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014*. The Internet Society, 2015. URL `http://www.internetsociety.org/doc/copperdroid-automatic-reconstruction-android-malware-behaviors`.

[15] Christoph Zauner. Implementation and benchmarking of perceptual image hash functions. 2010. URL `http://www.phash.org/docs/pubs/thesis_zauner.pdf`.

[16] Robert Tibshirani, Guenther Walther, and Trevor Hastie. Estimating the number of clusters in a data set via the gap statistic. *Royal Statistical Society*, 63, 2001. URL `https://web.stanford.edu/~hastie/Papers/gap.pdf`.

[17] TightVNC Team. TightVNC Software. URL `http://www.tightvnc.com/`.

[18] Google Inc. Android SDK, . URL `https://developer.android.com/sdk/index.html`.

[19] Xiaocong. uiautomator. URL `https://github.com/xiaocong/uiautomator`.

[20] Google Inc. Android Build Tools, . URL `https://developer.android.com/tools/revisions/build-tools.html`.

[21] OpenSSL comunity. Openssl. URL `https://www.openssl.org/`.

[22] National Institute of Standards and Technology. Recommended Elliptic Curves for federal government uses. July 1999. URL `http://csrc.nist.gov/groups/ST/toolkit/documents/dss/NISTReCur.pdf`.

[23] Google Inc. Google Play Store. URL `https://play.google.com/store/apps`.

[24] AppTown. Apptown Market. URL `http://www.apptown.nl/nl/`.

[25] 1Mobile. 1Mobile Market. URL `http://www.1mobile.com`.

[26] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pages 95–109. IEEE Computer Society, 2012. ISBN 978-0-7695-4681-0. doi: 10.1109/SP.2012.16. URL `http://dx.doi.org/10.1109/SP.2012.16`.

[27] Timothy Vidas and Nicolas Christin. Evading android runtime analysis via sandbox detection. In Shiho Moriai, Trent Jaeger, and Kouichi Sakurai, editors, *9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14, Kyoto, Japan - June 03 - 06, 2014*, pages 447–458. ACM, 2014. ISBN 978-1-4503-2800-5. doi: 10.1145/2590296.2590325. URL `http://doi.acm.org/10.1145/2590296.2590325`.