POLITECNICO DI MILANO
SCUOLA DI INGEGNERIA DELL'INFORMAZIONE
Corso di Laurea Specialistica in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

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

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

Tesi di Laurea di:
Andrea Gianazza (M. 755119)

Anno Accademico 2012-2013
Facoltà di Ingegneria dell'Informazione (Ing.V - INF)
INGEGNERIA INFORMATICA (187)
Laurea Specialistica (ord. 509)

Abstract

With the growing popularity of smartphones and apps, cyber criminals have begun infecting mobile devices with malware. The prevalence and complexity of such malicious programs are constantly on the rise.

In response, researchers proposed various approaches for analysis and detection of malicious applications. In this work we focus on the fundamental problem of malware analysis, which can be traditionally divided in two categories: static and dynamic analysis. Static approaches have the main advantage of high code coverage and scalability, but they are ineffective against code obfuscation or dynamic payloads. Conversely, dynamic approaches are resilient to code obfuscation but they reach lower code coverage than static ones. In particular, current solutions are unable to fully reproduce the typical usage of human users, so to only partially cover the malicious code during dynamic analysis. Consequently, a malware sample may not exhibit its malicious behavior, leading to an erroneous or incomplete analysis.

In this work we propose a new approach to exercise the user interface (UI) of an Android application in order to effectively exercise potentially malicious behaviors. To this end, our key intuition is to record and reproduce the typical UI-interaction of a potential victim of the malware, so to stimulate the relevant behaviors during dynamic analysis. Moreover, to make our approach scale, we record a trace of the UI events while a human is using an app, and then automatically re-execute this trace on apps that are "similar" to the given one. We developed our approach in PuppetDroid, an Android remote execution environment and UI exerciser for dynamic malware analysis.

We experimentally demonstrated that our stimulation approach allows to reach higher code coverage than automatic UI exercisers, so to succeed in stimulating interesting malicious behaviors that are not exposed when using other approaches.

Finally, PuppetDroid relies on crowdsourcing to collect new stimulation traces: we believe that our system can attract the interest not only of security analysts but also of normal users that want to safely test potentially malicious applications.

Sommario

Gli ultimi anni sono stati testimoni di una crescita esponenziale nell'adozione e diffusione di dispositivi mobili come smartphone e, più recentemente, tablet. In particolare, gli attuali smartphone non possono più essere etichettati come semplici dispositivi per fare chiamate telefoniche o mandare SMS, ma sono diventati complessi e potenti strumenti che permettono la navigazione web, l'utilizzo di social network, la navigazione tramite GPS e l'esecuzione di operazioni bancarie. A supportare ulteriormente la diffusione di questi dispositivi ha contribuito la comparsa di numerosi negozi virtuali che permettono agli utenti di poter accedere ed installare sui propri dispositivi una vasta gamma di applicazioni in maniera semplice ed immediata. Tutto questo ha portato, verso la fine del 2012, al definitivo sorpasso del numero di dispositivi mobili attivi rispetto al numero di PC convenzionali.

La crescente popolarità di smartphone e delle relative applicazioni ha inevitabilmente attratto l'attenzione di criminali informatici, che hanno quindi iniziato a sviluppare applicazioni malevoli per dispositivi mobili. Un'applicazione malevole, o *malware*, è un'applicazione sviluppata col chiaro intento di compiere delle azioni all'insaputa dell'utente che possono spaziare dal furto di informazioni sensibili, alla sottoscrizione a servizi a pagamento o all'esecuzione di operazioni con l'intento di rendere il dispositivo inutilizzabile. Come mostrato inoltre da recenti report di produttori di anti-virus ed esperti del settore, la diffusione e la complessità dei malware è in continua crescita e Android risulta la piattaforma più colpita.

In risposta a questo fenomeno, sono state proposte diverse tecniche per analizzare ed individuare nuove minacce. In questo lavoro ci focalizziamo sull'analisi di malware. In particolare, i due principali approcci adottati per lo studio di nuovi malware sono l'analisi statica e quella dinamica. Le tecniche di analisi statica prevedono l'utilizzo di strumenti di decifratura, decompilazione o disassemblaggio per analizzare il codice di un'applicazione a vari livelli. Per esempio, un tipico scenario di utilizzo è la ricerca all'interno del codice di schemi di esecuzione tipici, detti *signature*, per poter individuare velocemente se un'applicazione può essere etichettata come malevola oppure no: questa tecnica è solitamente alla base del funzionamento della

maggior parte degli anti-virus. Gli strumenti di analisi statica presentano il vantaggio di raggiungere un'alta copertura di codice, poiché permettono di analizzare percorsi di esecuzione di difficile accesso, e un'alta scalabilità, perché l'analisi può essere automatizzata in maniera particolarmente agevole. Questi strumenti risultano però spesso inefficaci in presenza di codice offuscato, ovvero codice appositamente modificato per renderne difficile la lettura e l'analisi, e payload dinamici, ovvero porzioni di codice che vengono recuperate dal programma durante la sua esecuzione e quindi non accessibili staticamente. Le tecniche di analisi dinamica prevedono invece di studiare il comportamento di un programma, potenzialmente pericoloso, eseguendolo in un ambiente sicuro e controllato, definito *sandbox*. Tipiche soluzioni adottate in questo approccio prevedono l'utilizzo di *network sniffer*, ovvero strumenti in grado di analizzare e tenere traccia del traffico di rete, o di tecniche di istrumentazione del codice, per tracciare le chiamate di sistema invocate dall'applicazione. Queste informazioni vengono poi utilizzate dall'analista per risalire ai comportamenti malevoli, definiti *malicious behavior*, messi in atto dal malware. L'analisi dinamica non risente della presenza di codice offuscato, dal momento che prima o poi questo codice deve "de-offuscarsi" durante l'esecuzione, o di dynamic payload, però difficilmente riesce ad ottenere l'alta copertura di codice o la scalabilità tipica dell'analisi statica. La copertura di codice è in particolare la limitazione principale di questo approccio: infatti l'analisi dinamica può studiare le azioni eseguite in un percorso di esecuzione solamente se quel percorso viene effettivamente esplorato. Le soluzioni attualmente disponibili cercano di superare questa limitazione utilizzando strumenti di stress-testing o sistemi che combinano l'uso di tecniche di analisi statica con stimolatori di *User Interface* (UI) automatici: il problema di questi approcci è che difficilmente riescono a ricreare l'interazione tipica di un utente umano, generando quindi una stimolazione spesso inefficace e una scarsa copertura di codice.

L'obbiettivo di questo lavoro è di proporre un approccio innovativo per condurre test di analisi dinamica, definendo un metodo in grado di stimolare in maniera efficace l'interfaccia di un'applicazione. L'approccio che proponiamo ruota intorno a due concetti chiave. Il primo è quello di sfruttare la stimolazione fornita da un tester umano per riprodurre la tipica interazione generata da una vittima del malware: a differenza infatti degli stimolatori automatici, l'utente umano comprende la semantica degli elementi visualizzati a schermo ed è in grado di stimolare l'applicazione in maniera corretta. Il secondo concetto chiave è quello di registrare una traccia dell'interazione dell'utente con l'applicazione e sfruttarla per stimolare applicazioni che presentino un layout simile a quello dell'applicazione originariamente testata. La nostra idea è avvalorata da una pratica particolarmente diffusa tra gli autori di malware, ovvero il repackaging di malware esistenti per creare delle varianti in grado di non essere individuate dagli anti-virus. Tramite il nostro

approccio quindi, se almeno un utente riesce a stimolare un comportamento malevolo in un malware, è molto probabile che rieseguendo quella traccia di interazione su applicazioni simili, sia possibile stimolare comportamenti malevoli simili.

Basandoci su queste osservazioni, abbiamo implementato il nostro approccio in PuppetDroid, un ambiente di esecuzione remota per l'analisi dinamica di malware Android. PuppetDroid mette a disposizione dell'utente 2 metodi per eseguire l'analisi dinamica di un'applicazione (malevola):

1. **Test Manuale**: l'utente interagisce direttamente con l'applicazione ed esegue il test utilizzando la sandbox che mettiamo a disposizione.
2. **Test Automatico**: l'utente sfrutta tracce di stimolazione, registrate da test manuali precedentemente eseguiti, per stimolare in maniera automatica la UI di applicazioni simili.

I test manuali permettono agli utenti di testare applicazioni potenzialmente pericolose tramite il proprio smartphone, senza alcun rischio di infezione o furto di dati sensibili. L'applicazione viene infatti eseguita su una sandbox remota opportunamente istrumentata per poter effettuare analisi dinamica. Inoltre l'interazione degli utenti con le applicazioni vengono registrate in tracce di stimolazione: queste tracce vengono usate nei test automatici per stimolare applicazioni simili in maniera automatica, rendendo quindi il nostro approccio scalabile. Infine, il nostro sistema può fare affidamento sul crowdsourcing per la raccolta di nuove tracce di stimolazione: riteniamo infatti che PuppetDroid possa attirare l'attenzione non solo di esperti di sicurezza informatica, ma anche di utenti comuni che vogliono provare in maniera sicura delle applicazioni che hanno scaricato dal Web o da negozi alternativi.

I risultati dei test sperimentali condotti per valutare il nostro approccio mostrano che sia i test manuali che i test automatici, eseguiti riutilizzando tracce di stimolazione precedentemente registrate, permettono di ottenere una copertura maggiore, in termini di behavior stimolati, rispetto a quella ottenuta con i classici strumenti di stimolazione automatica. Inoltre, abbiamo individuato anche alcuni casi particolari in cui PuppetDroid riesce a stimolare behavior malevoli, che non sono invece esposti utilizzando approcci di stimolazione differenti.

I contributi originali presentati in questa tesi possono essere riassunti nei seguenti punti:

- Proponiamo un nuovo approccio per l'analisi dinamica di malware per Android che sfrutta la stimolazione eseguita da un tester umano in modo da ottenere una maggiore copertura di codice.
- Proponiamo un metodo innovativo per stimolare in maniera automati-

ca la UI di un'applicazione, riutilizzando tracce di stimolazione eseguite su applicazioni simili precedentemente analizzate.

– Abbiamo implementato il nostro approccio in PuppetDroid, un servizio di semplice utilizzo che sfrutta un ambiente di esecuzione remota per permettere agli utenti di testare in maniera sicura applicazioni potenzialmente pericolose, senza alcun rischio di infezione o furto di dati sensibili.

– Abbiamo valutato sperimentalmente il nostro approccio e dimostrato che sia la stimolazione manuale che quella automatica, ottenuta tramite il riuso di tracce di stimolazione, permettono di ottenere una maggiore copertura di codice rispetto a quella ottenuta con altri approcci di stimolazione automatica.

Contents

List of figures

List of tables

List of listings

Chapter 1

Introduction

In recent years, we witnessed an explosive growth in smartphone sales and adoption: as reported by Canalys [4], in 2011 the number of smartphones sold worldwide surpassed the number of conventional PCs, and the total number of active units surpassed the 1 billion mark in the third quarter of 2012 [3]. The popularity of smartphones can be partially attributed to the particular functionality and convenience they offer to end users. In fact, existing mobile phones are not simply devices used to make phone calls and receive SMS messages, but powerful communication and entertainment platforms for web surfing, social networking, GPS navigation, and online banking. Furthermore, their usability has been greatly supported by the emergence of many app marketplaces, online stores where users can browse through different categories and genres of applications and automatically download and install them on their devices. Ubiquity and ease of handling of such devices made them the new personal computers.

Unfortunately, such popularity also attracted the attention of cyber criminals: as a matter of fact, security reports by *Anti-Virus* (AV) vendors [42, 40, 8, 30] illustrate a notable increase in the number of threats against mobile platforms during the last year. In particular, these reports point out that Android appeared to be the operating systems most targeted by mobile threats in 2012.

The distribution of malicious software, commonly known as *"malware"* or *"mobile malware"*, for smartphones is certainly the main threat. In response, researchers proposed various approaches for analysis and detection of malicious applications. In this work we focus on the fundamental problem of malware analysis, which can be traditionally tackled with two types of program-analysis techniques: static and dynamic analysis. Static approach analyzes the code at various levels (e.g., machine, assembly or source). Static analysis is used, for instance, to find known patterns of malicious code or

to reconstruct a program's *Control Flow Graph* (CFG). Static analysis has the main advantage of high code coverage and scalability, but it is ineffective against *obfuscated code* (i.e., source or machine code properly modified in order to make it difficult to be read and analyzed) or *dynamic payloads* (i.e., portion of code dynamically downloaded during program execution). On the other hand, dynamic analysis studies the behavior of a program by observing its execution in a *sandboxed environment* (i.e., a controlled environment expressly created to execute potentially dangerous applications in a safe way). A malicious behavior identifies any possible action performed by the application against the will of the user and it can range from sensitive information leaking, to the subscription to premium-rate services or the disruption of user's device. Typical dynamic analysis techniques involve the use of packet analyzers to intercept and log network traffic generated by the malware or the use of instrumentation techniques in order to keep trace of system calls invoked by a malicious application. This information is used by analysts to examine the actions performed by an unknown program during its execution and to reconstruct its behavior. Dynamic analysis is not circumvented by code obfuscation because code has to deobfuscate itself sooner or later during execution. Eventually, the malware will exhibit its malicious behavior. However, the main limitation of dynamic analysis is its inability to reach high code coverage: dynamic analysis can examine the actions performed in an execution path only if that path is actually explored. Current solutions mitigate this limitation leveraging stress-test tools or a combination of static analysis and automatic *User Interface* (UI) exercisers: the problem of these approaches is that they are often not able to reproduce the typical usage of a human user, providing in this way an ineffective stimulation and unsatisfactory code coverage. Exercising mobile applications in a proper way is not trivial problem because mobile applications make use of highly-interactive UIs in order to leverage the capabilities of modern touchscreen devices.

In this work we propose a new approach to exercise the UI of an Android application in order to change the way malware analysis experiments are conducted and effectively exercise potentially malicious behaviors. To this end, our key intuition is to analyze and reproduce the typical UI-interaction of a potential victim of the malware, stimulating in this way relevant behaviors for analysis purposes. Unlike automatic exercisers, human user understands the semantic of the UI elements and exercises the application accordingly. A quite common practice among mobile malware authors is to repackage already existent malware samples, inserting small changes that allow them to obtain new samples able to avoid detection techniques with minimum effort. Our idea is to record a trace of the human-driven UI stimulation performed during a test and automatically re-execute this trace on applications similar to the one originally tested by the user. In this way, if at least one user in our system succeeds in manually stimulating a malicious behavior in a malware,

it is quite likely that by re-using UI stimulation traces, we can stimulate similar malicious behaviors.

Based on the above observation and intuitions, in this work we propose two ways to perform dynamic analysis of a (malicious) Android application:

1. **Manual Test**: the user directly interacts with the application and executes the test leveraging the sandboxed environment we provide.
2. **Automatic Test**: the user leverages previously recorded UI stimulation traces performed on similar samples to automatically exercise the application.

Manual testing allows us to exercise in an effective way the UI of the application. Moreover, we record the interaction of the user with the application and we store it in a *stimulation trace*: with this term we indicate the sequence of actions performed by the user and the list of UI elements actually stimulated during the test. We then leverage these stimulation traces to support automatic testing: in this way, we can make our approach scale and effectively test repackaged variants of malware previously tested by human users. Finally, we can rely on crowd-sourcing to collect new stimulation traces: we believe that our system can attract the interest not only of security analysts but also of normal users that want to safely try potentially malicious applications they found on the web or in alternative markets.

We developed our approach in PuppetDroid, an Android remote execution environment and UI exerciser for dynamic malware analysis. Our system let security analysts perform manual tests on potentially malicious Android applications using their personal devices, avoiding any possible risk of infection or information leaking to them. The application actually executes on a remote sandbox. The typical usage scenario of PuppetDroid is as follows. The security analyst uploads a suspicious application to our service. If a stimulation trace is available, PuppetDroid uses it to exercise the application, which runs in the (instrumented) Android emulator. Upon test termination, the system returns recorded behaviors to the analyst. If no stimulation trace is available, the analyst is prompted to either provide a stimulation trace (i.e., by exercising the application from his or her smartphone) or to enqueue the application for analysis. In this latter case, whenever a new turk (i.e., a human worker in the crowdsourcing terminology) is available, he or she provides the stimulation trace for the enqueued application.

We experimentally evaluated our approach. The results showed how both manual exercising and re-execution of stimulation traces allow to reach higher code coverage than the one obtained with automatic UI exercisers: as a matter of fact, we succeeded in stimulating more than twice the number of behaviors stimulated by other exercising strategies. Moreover, we found some particular cases in which PuppetDroid succeeds in stimulating inter-

esting malicious behaviors that are not exposed using automatic application exercising approaches.

The original contributions presented in this thesis can be summarized as follows:

- We propose a new approach to Android dynamic malware analysis that leverages human-driven UI stimulation to increase code coverage.
- We propose an original method to automatically exercise the UI of an unknown application re-using UI stimulation traces obtained from previously analyzed applications that present a similar layout.
- We implemented our approach in PuppetDroid, an easy-to-use service that leverages remote sandboxing to allow users to safely test potentially malicious applications without any risk of infection or information leakage.
- We experimentally evaluated our approach demonstrating that manual exercising and stimulation trace re-execution allow to reach higher code coverage than the one obtained with automatic UI exercisers.

The document is organized as follows.

In *Chapter 2* we introduce the Android operating system, we describe the problem of mobile malware and present the most relevant works in the fields PuppetDroid deals with. We then analyze the current limitations of existing Android malware analysis systems and show how our approach can bring an innovative contribution to the state of the art.

In *Chapter 3* we explain the design and implementation choices we made to build PuppetDroid. We initially present an overview of the system, subsequently describing each component in detail. Here we also present puppet_rerunner, the tool we implemented to re-execute previously recorded UI stimulation on repackaged applications.

In *Chapter 4* we describe the experiments we made to evaluate our approach. Specifically, we compare our human-driven stimulation approach with currently used automatic stimulation strategies. Then we propose a set of experiments to evaluate the effectiveness of our re-execution method.

In *Chapter 5* we analyze the main limitations of PuppetDroid system, focusing on performance and the problem of evasion from dynamic analysis environments.

In *Chapter 5* we overview relevant works in two fields related to this thesis: static malware analysis and Android applications similarity.

Last, in *Chapter 7* we draw the conclusions of our work, discussing the original contributions that we provided and suggesting future developments and

improvements to PuppetDroid system.

Chapter 2

Background and state of the art

In this chapter we introduce the Android platform, presenting a general overview of the operating system and then focusing on security aspects (*Section 2.1*). In *Section 2.2*, we present the state of the art in the main fields this thesis deals with: malware analysis and UI stimulation. Finally, in *Section 2.3* we illustrate the limitations of existing Android malware analysis techniques and introduces our approach to face them.

2.1 The Android platform

Android is a popular open-source operating system primarily designed for touchscreen mobile devices, such as smartphones and tablet computers, and developed and supported by a group of companies known as the Open Handset Alliance, led by Google [23]. Publicly unveiled in 2007, Android has seen an explosive proliferation during the last years. Eric Schmidt, Google's executive chairman and former CEO, has recently announced [33] that his company is now seeing 1.5 million activations per day, boasting an installed base of about 750 million users. Thanks to these numbers, Android nearly reached 80.0% of the mobile market share [22] and has an upward trend with a growth of 73.5% in 2013.

To simplify the application distribution, discovering, purchasing and updating, the last years have seen the emergence of many app marketplaces, online stores where users can browse through different categories and genres of applications, view information and reviews of them, and automatically download and install the application files on their devices. Indeed, along with the number of Android devices, the prevalence of the official app market, Google

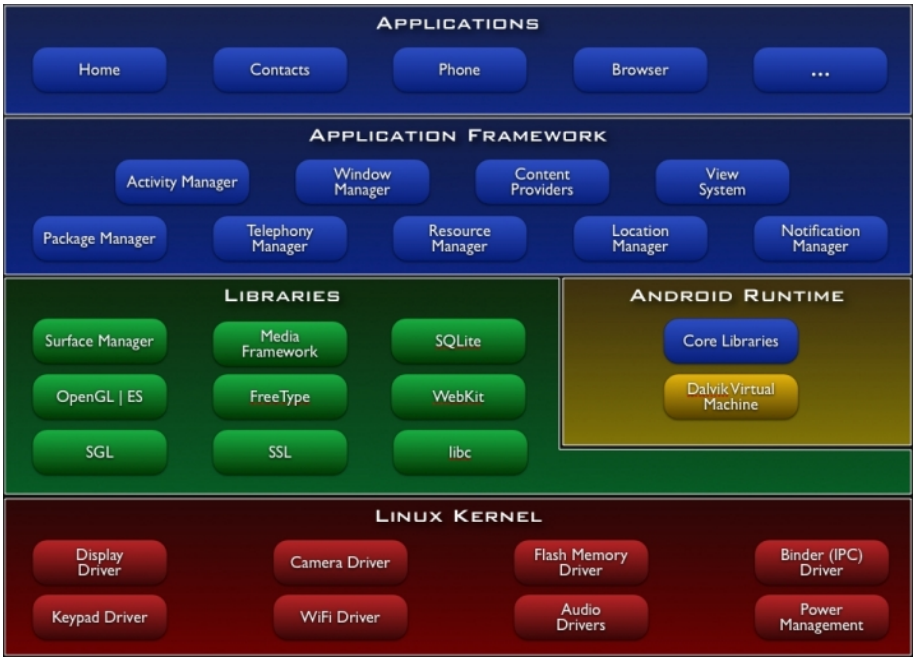

Figure 2.1: Android software stack.

Play Store, increased this year hitting 48 billion downloads and over 1 million available applications, as stated by Sundar Pichai, senior vice president at Google, during Google I/O 2013 [43]. In addition to the official market, tens of unofficial/alternative markets (e.g., Amazon App-Shop, Samsung Apps, AndroLibs, AppBrain), offer exclusive applications, or applications that were banned from the official repository.

2.1.1 Overview

The core operating system is built on top of the Linux kernel, which provides a hardware abstraction layer to make Android easily portable to a variety of platforms. Most of the core functionality is implemented as applications running on top of a customized middleware. The middleware itself is written in Java and C/C++. **Figure 2.1** shows the various level of the Android software stack.

Applications are most often written in Java and compiled to a custom byte-code known as the *Dalvik EXecutable* (DEX) bytecode format. In particular, during the compilation process, a Java compiler first generates the Java byte-code, as a set of `.class` files, and then the Dalvik `dx` compiler consumes the `.class` files, recompiles them to Dalvik bytecode and finally writes the resulting application into a single `.dex` file. However, all applications can

contain both Java and native components. Native components are simply shared libraries that are dynamically loaded at runtime: the *Java Native Interface* (JNI) is used to allow communications between the native and Java code. The *Dalvik Virtual Machine* (DVM) is then used to provide a Java-level abstraction in order to provide a sandboxed application execution environment. An Android application is distributed using the *Android Package* (APK) format, that is basically a ZIP formatted file containing the above mentioned `classes.dex`, an XML manifest file that describes the application and finally libraries and resources used by the application.

2.1.2 Android security

The Android platform takes advantage of the Linux user-based protection as a means of identifying and isolating application resources. As a matter of fact, each application runs in a process with low-privilege user ID, assigned at installation time, and applications can access only to their own files. By default, an Android application can only access a limited range of system resources. The system restricts Android application access to sensitive resources, as telephony and SMS/MMS functions, network/data connections, geolocation, etc. In order to implement these restrictions, the Android framework provides a set of sensitive APIs that are intended for use only by trusted applications and protected through a permission mechanism. According to this mechanism, before installing a new application on the device, the system displays a screen listing the permissions required by the application. The user has to accept them in order to complete the installation.

To enforce permissions, various parts of the system invoke a validation mechanism to check whether a given application has a specified permission. There is no centralized policy for checking permissions when an API is invoked. Rather, the API implementation calls the validation mechanism to check that the invoking application has the necessary permissions. A small number of permissions (e.g., `INTERNET` or `BLUETOOTH`) are enforced using Unix groups ID control policy.

This permission mechanism is coarse-grained, because the user cannot grant or deny individual permission. He or she must either grant or deny the whole requested permissions. Furthermore, once granted, the permissions are applied to the application as long as it is installed and there is no mechanism to re-adjust the permissions or constrain the runtime behavior of the application later on. Some recent works try to address this problem by modifying the original Android operating system with finer security modules.

For instance, TISSA [51] is a privacy module that allows users to have a fine-grained control on which kind of user information can be accessible by an

untrusted application. TISSA adds lightweight control modules to the Android framework and it is orthogonal to the Android permissions mechanism. In particular, after the installation, when an untrusted application tries to access sensitive data, the user has the possibility to choose if to provide correct, empty, anonymized or bogus information: in this way the user can prevent personal information leakage without affecting the normal behavior of the application.

However the approach shown in TISSA relies on modification of the underlining software stack, which makes the deployment to off-the-shelf Android phones difficult. This problem is addressed in AppGuard [1], a practical extension of the Android permission system that allows to dynamically control enforced permissions and to apply fine-grained security policies, without requiring modifications of the firmware (i.e., the software embedded in the device) or *root* access to the smartphone. The basic idea of AppGuard is to rewrite an untrusted application such that the code that monitors the application is directly embedded into its bytecode. In this way AppGuard is able to observe a trace of security-relevant events, injecting security checks at critical points into the application code, and then to enforce custom security policy, suppressing or altering calls to security-relevant methods. As mentioned above, third party applications installed on Android are assigned distinct user ids and, by default, an application can neither access nor modify another application: therefore, since AppGuard cannot simply modify already installed applications, it has to repackage and reinstall the untrusted application.

A recent study [9] analyzes the Android permission system focusing on the case of over-privileged applications: this work shows how the lack of reliable permission information in the official documentation generate confusion and lead the developers to ask for unnecessary permissions to make their application work correctly. The situation for end users is even more complicated, because they can hardly understand the proper semantics of the permissions required at installation time: this fact makes them unaware of possible security and privacy impact of their decision.

The Android permission mechanism is addressed also in Woodpecker [18]: this work highlights how a bad enforcement of the Android permission model could lead to capability leaks (i.e., the situations where an application can gain access to a permission without actually requesting it). The authors distinguish two categories of capability leaks: *Explicit capability leaks*, which exploit some publicly accessible interfaces or services to access certain permissions, and *Implicit capability leaks*, which allows to an application to inherit permissions from another application with the same signing key, subverting in this way the permission-based security model. The first capability leak happens when a privileged application allow any caller to invoke its

entry points, without checking the caller's credentials before it accesses to dangerous functionalities. An implicit capability leak happens when an application has the attribute `sharedUserId` in its manifest but does not require a dangerous permission: in this case is possible that another application, that has both that specific permission and the same user id of the previous one, is already installed on the smartphone. If it is the case, the first application can use the same permissions of the second one without explicitly requesting them.

2.1.3 Android malware

Nowadays, people use smartphones for many of the same purposes as desktop computers, most notably web browsing, social networking and online banking. Moreover, smartphones provide features that are unique to mobile phones, like SMS messaging, constantly-updated location data and ubiquitous access. As a result mobile devices have become appealing for cyber criminals, whose current activities include the infection of mobile devices with the final goal of stealing sensitive data, or performing fraudulent monetary transactions without the user's consent. Recent reports published by *Anti-Virus* (AV) vendors [27, 40, 8, 42, 30] show the importance of this trend, as illustrated in **Figures 2.2** and **2.3**.

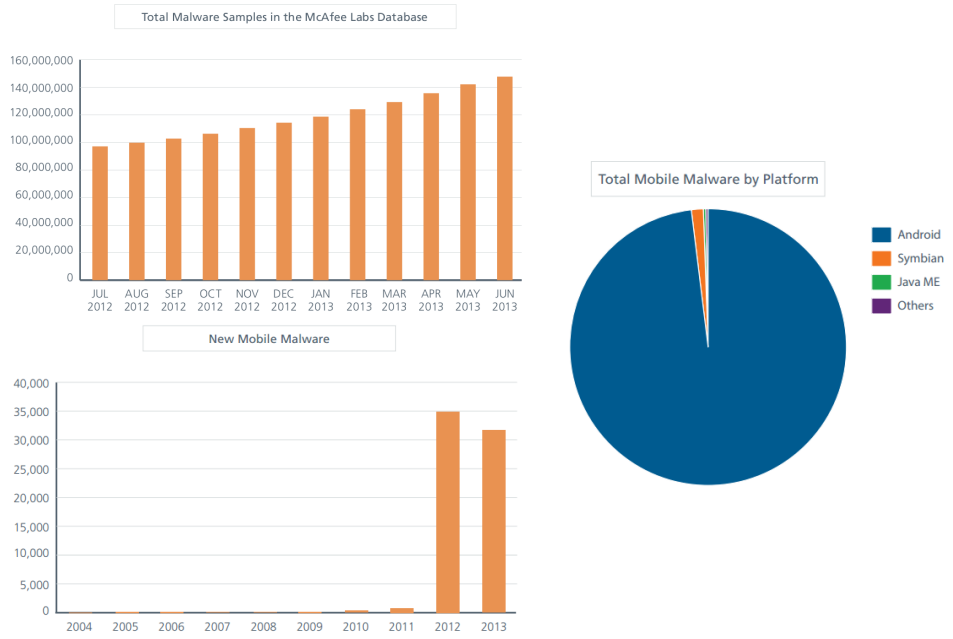

Figure 2.2: Android malware growth - McAfee 2013 Q2 Threats Report [30].

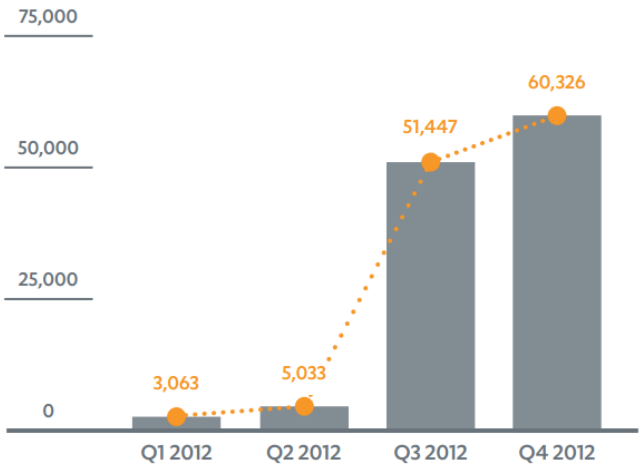

Figure 2.3: Number of Android threats - F-Secure 2012 Mobile Threat Report [8].

These reports also state that the mobile malware industry is mature and become a viable business for attackers, as shown in **Figure 2.4**.

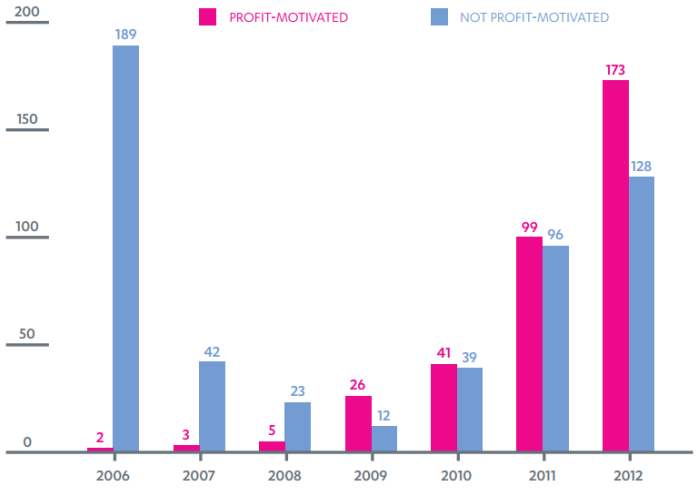

Figure 2.4: Mobile threats motivated by profit per year, 2006-2012 - F-Secure 2012 Mobile Threat Report [8].

Another interesting study is by Zhou and Jiang [50], who present the first scientific, thorough and comprehensive characterization of the Android mobile malware phenomenon. The authors collected more than 1,200 malware samples by crawling a variety of Android markets, both manually and automatically, between August 2010 and October 2011. They also publicly released the resulting dataset [48].

This work presents an interesting characterization of Android malware based on the installation method used. In particular, there are three main methods:

Repackaging – this is the most common technique used to hide malicious payloads into an application: in essence, malware authors locate and download popular apps, disassemble them, enclose malicious payloads and finally re-assemble and submit the new application to official and/or alternative Android markets[1]. This process is also shown in **Figure 2.5**, taken from [2];

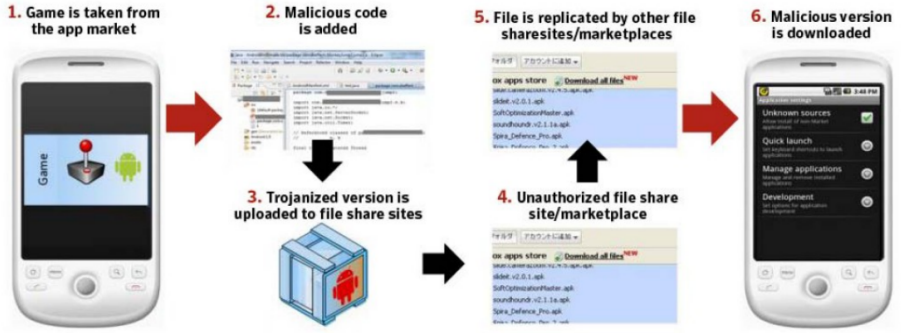

Figure 2.5: Workflow of the repackaging technique - ITU Regional forum on Cybersecurity [2].

Update Attack – in this technique, the malicious payloads are downloaded at runtime. In this way the application has only to include an update component and, as a result, it is more difficult to detect it through static scanning;

Drive-by Download – it is the traditional drive-by download attack, ported to mobile space, where the user is enticed through social-engineering tricks to download "interesting" or "feature-rich" applications.

The authors present another interesting characterization, this time dividing the malicious payload functionalities in 4 categories:

Privilege Escalation – the payload tries to exploit known platform-level vulnerabilities in order to gain root privileges;

Remote Control – the infected phone is transformed in a bot in order to be remotely controlled by a C&C server;

Financial Charge – the malware stealthily subscribes the victim to attacker-controlled premium-rate services, such as by sending SMS messages as shown in **Figure 2.6**;

[1] A non-exhaustive list of alternative Android market is published here: `http://blog.andrototal.org/post/53390790840/unofficial-android-marketplaces`

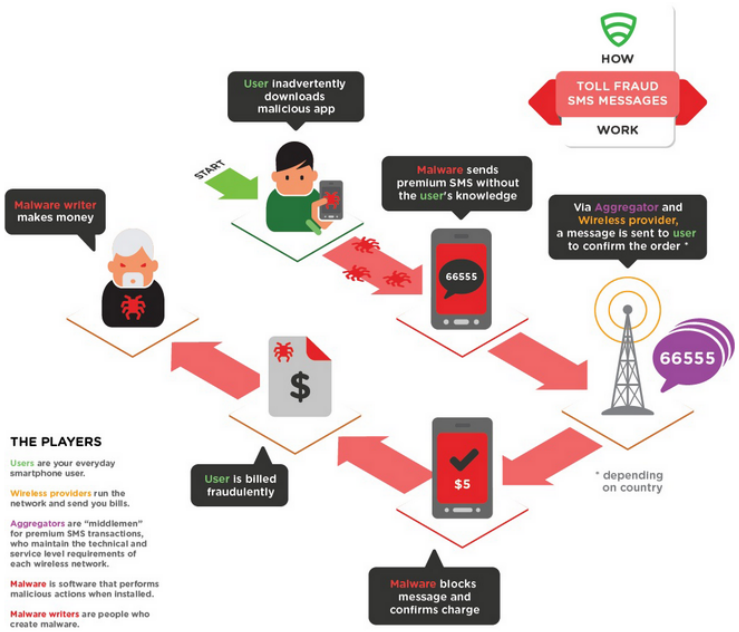

Figure 2.6: How Toll Fraud SMS Messages works - Lookout 2012 Mobile Security Report [27].

Information Collection – the malware harvests various information from the infected phone, such a SMS messages, phone numbers and user accounts. To make an example, Zitmo, the Zeus version on Android, and Spitmo, the SpyEye version on Android, intercept SMS verification messages in order to generate fraudulent transactions on behalf of infected users.

Based on this analysis, the authors proposed a taxonomy that includes 49 distinct families. An in-depth analysis of the evolution of two families, namely DroidKungFu and AnserverBot, further analyzed by the same authors in [49], showed that their complexity increased over time. A retrospective analysis of the variants diversity, published in [7], corroborates this result: as summarized in **Figure 2.7**, not only the complexity of malware increased again last year, but the number of variants also increased.

Some interesting examples demonstrating the increase of malware complexity came out during the past year. SMSZombie, for instance, is a malware first discovered around July 2012. It allegedly infected more than 500,000 smartphones in China and leveraged a vulnerability in the China Mobile SMS Payment system process in order to generate unauthorized payments, steal bank card numbers and gather other financial information. SMSZombie pretends to be a clean wallpaper application, which instead stealthily carries a malicious payload as a second APK file disguised as a JPEG asset. The

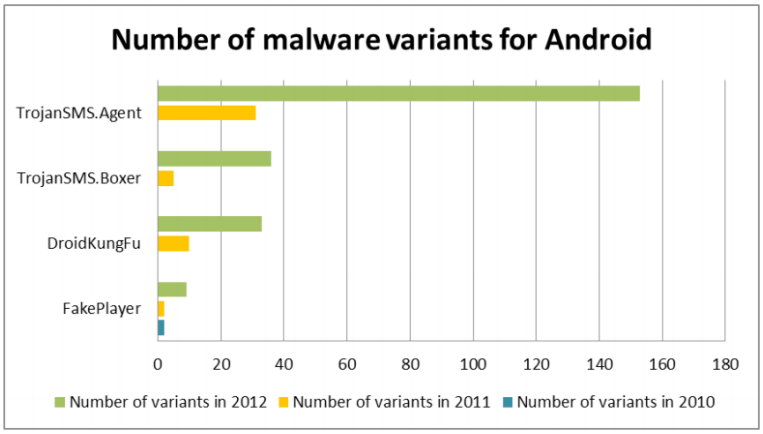

Figure 2.7: Android malware variants growth - ESET Trends for 2013 [7]).

trojan pushes user to install some additional files that included the actual malicious code. Once the malicious package is installed, the user is forced to register it as a *Device Administration Application*, making its removal particularly tricky. The malware also enforces evasion tricks in order to not be detected. The only way to remove SMSZombie consists in a manual operation (i.e., Device Administration removal), which cannot be performed by any antivirus application.

Another notable malware is FakeInstaller, also referred to as RuSMSMarket, OpFake, Fakebrows, and FakeWAM: it is by far the most detected malware by AV products in 2012, according to security reports published by AV vendors [27, 42, 32]. FakeInstaller sends SMS messages to premium rate numbers, without the user's consent. This malware pretends to be the installer for a legitimate popular application, such as *Opera Browser* (hence the names OpFake and Fakebrows) or *WhatsApp Messenger*. The victim usually retrieves it from fake official sites or from fake markets found through search engines or social networks. These sites fool the user making the applications appear to be legitimate, including screenshots, descriptions, user reviews, videos, etc. There are many variants of this malware distributed on hundreds of websites and fake markets. Furthermore, it uses server-side polymorphism, obfuscation, anti-reversing techniques and frequent recompilation, all to avoid detection by antivirus solutions. A detailed description of this malware can be retrieved here [32].

Finally, an interesting survey on mobile malware is presented in [10]. The authors collected and analyzed malware samples for Android, iOS and Symbian in order to categorize them and extract information about the motivations underlying malware propagation. Furthermore, the three frameworks are compared according to enforced permission mechanisms and adopted

reviews process in order to prevent the spread of malware.

2.2 State of the art

In this section we overview and compare the state of the art in the fields addressed by PuppetDroid system. In particular in *Section 2.2.1* we illustrate malware analysis and present existing analysis tools for the Android platform, while in *Section 2.2.2* we present the main approaches used to stimulate modern touchscreen *User Interface*s.

2.2.1 Malware analysis techniques

Program analysis indicates the whole set of processes and techniques used to reconstruct and analyze the behavior of an unknown program: software developers usually leverage it for different purposes (e.g., to verify the correctness of their programs, to find possible flaws in the code or to test performance during program execution). Malware analysis is a specific use-case of program analysis, aimed to analyze the behavior of malicious applications: it is a fundamental field in computer security because it allows to study the execution patterns commonly exploited by malware authors, allowing in this way to build robust and up-to-date security products.

Program analysis approaches can be divided in two categories: static and dynamic analysis.

Static analysis involves different investigation techniques that analyze the code at various levels (e.g., machine, assembly or source) and allow to examine a program without executing it. These techniques usually include decompilation, disassembling, decryption, control and data flow analysis and others. A typical use-case of this approach is the discovering of sections of code that follows known malicious pattern of execution, named *"signatures"*, and is at the base of *Anti-Virus* (AV) products.

The main advantage of static analysis is that it can potentially see the whole behavior of a program since it can examine parts of a program that normally do not execute, reaching in this way a complete code coverage analysis. Moreover static analysis is very fast and can easily scale, allowing the analysis of a huge number of applications in a relative small amount of time.

The main disadvantage of this approach is its inability to manage *obfuscated code* (i.e., source or machine code properly modified in order to make it difficult to be read and analyzed) or *dynamic payloads* (i.e., portion of code dynamically downloaded during program execution), two things particularly

present in malware samples. In particular, as previously mentioned, *Upload Attack* and *Drive-by Download* are two techniques often used by malware authors to let an application download malicious payloads at run-time: in this case a static analysis approach cannot know the behavior of these payloads and eventually fails.

On the other hand, dynamic analysis studies the behavior of a program executing it: usually dynamic analysis tools include debuggers, function call tracers, network sniffers, machine emulators and others. Typical examples of dynamic analysis are the use of packet analyzers to intercept and log network traffic generated by the malware or the use of instrumentation techniques in order to keep trace of system calls invoked by the malicious application. A common approach to dynamic analysis involves the execution of a program in a *sandboxed environment* (i.e., a controlled environment expressly created for running unknown software in a safe way).

Dynamic analysis often let the analyst to better understand the behavior of an unknown program and it succeeds in overcoming the limitations shown by static analysis. For instance, since dynamic payloads are download and executed at runtime, dynamic analysis is able to examine also these portions of code. Moreover, instrumentation techniques let dynamic analysis be resilient against code obfuscation: in fact, code has to deobfuscate itself sooner or later during execution and, eventually, the malware will exhibit its malicious behavior.

The clear disadvantage of this approach is that it can only see the executed paths: hence, if the malicious behavior is hidden in a path not executed during the analysis, it is not detected. Other disadvantages of dynamic analysis are that it requires some human-driven or automatic strategy to exercise the UI of a program and it usually requires more time to examine an application in respect to static analysis. Finally, dynamic analysis often involves the use of instrumentation at user or kernel level that makes this approach detectable by the program under analysis.

As said, both these analysis approaches present advantages and drawbacks: then, the best practice to perform a complete and effective malware analysis involves the usage of a combination of these techniques.

In this work we focus on dynamic analysis, then we present here the state of the art in this field. Static analysis will be discussed in *Section 6.1*.

Dynamic malware analysis

One of the most used dynamic analysis framework of the last years is surely TaintDroid [6]. This analysis system has been developed to face the problem of private information management in mobile-phone systems, with particular

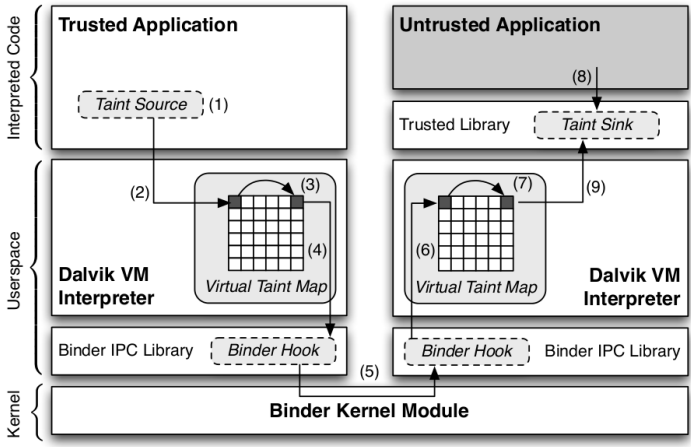

Figure 2.8: TaintDroid architecture.

reference to Android. As previously mentioned, Android provide only a coarse-grained permission system that allows to establish if an application can access to sensitive data, but it do not provide any insight on how this data is actually used. TaintDroid is an extension of Android that allows to keep trace of the flow of privacy sensitive data through third-party applications. The authors instrumented Android original code in order to label all system resources that try to take access to privacy-sensitive sources and notify the user in case there is the possibility that sensitive information can leave the system using the network, SMS messaging systems or other ways. **Figure 2.8** presents an overview of TaintDroid architecture.

The main limitation of TaintDroid is that it modifies the native library loader to ensure that third-party applications can only load native libraries from the firmware: this means that applications that use custom native libraries cannot work on TaintDroid. Another limitation of this system is that it is based on the manual instrumentation of the Android source code: this means that each version of TaintDroid can work only on the targeted Android version. Originally TaintDroid has been released for Android version 2.1 and 2.3, but recently a new version for Android 4.1 has been released.

Even if it had some limitations, TaintDroid has been one of the first really usable dynamic malware analysis framework for Android. For this reason it has been integrated and extended by other analysis systems: the two most famous system based on TaintDroid are DroidBox[2] and Andrubis [25].

DroidBox extended TaintDroid in order to add to the original taint-tracking system the ability to keep trace and log information related to network traffic, SMS sent, phone calls, file operations, cryptographic operations and started

[2]DroidBox project page: `https://code.google.com/p/droidbox/`

services. Since it is based on TaintDroid, this project brings with it all the limitations previously listed: moreover this project is no more supported and the last version addresses Android 2.3. Last year the authors of DroidBox released a new tool for dynamic analysis, named APIMonitor: instead of instrumenting Android source code, this new tool instruments the source code of the application under analysis. Even if this allows to untie the tool from a particular version of Android, there is the clear drawback that the original application must be modified and repackaged: this means that the malware author only needs to insert a signature check to make his/her application able to avoid this analysis.

Andrubis is the Android extension of Anubis[3], the famous Windows malware analysis service provided by International Secure Systems Lab. Andrubis leverages TaintDroid and DroidBox to perform automatic dynamic malware analysis of Android samples uploaded on its platform: in order to automatically stimulate Android applications, Andrubis leverages Monkey, a tool contained in the Android *Software Development Kit* (SDK) that will be described in *Section A.3*. Moreover, Andrubis makes use of apktool[4] and Androguard[5], two widely used tools for Android application reversing, to perform basic static analysis on the APK sample.

Another recently released framework for automatic dynamic malware analysis that leverages TaintDroid is AppsPlayground [36]. This framework integrates TaintDroid taint analysis with other dynamic analysis techniques in order to detect anomalous behaviors. Moreover, it provides a testing environment as similar as possible to the one of a real device, in order to mitigate evasion attempts. To automatically stimulate Android applications, AppsPlayground analyzes the information displayed on the screen in order to reconstruct application's logic and perform intelligent execution.

All the solutions presented so far exploit the instrumentation of Android source code in order to perform dynamic analysis. The main advantage of this approach is that a real device can be used as sandbox to host dynamic analysis testing, eliminating in this way performance limitations related to the use of an emulated environment. On the other hand, this type of instrumentation has a clear drawback: if a malware succeeds in obtaining root privileges, it can detect the presence of the analysis framework and hide its malicious behavior.

To overcome this limitation, all the analysis components must be at a more privileged level in respect to Android system environment. The only way to do that is to perform out-of-the-box dynamic analysis, instrumenting the

[3]Anubis project page: `http://anubis.iseclab.org/`
[4]apktool project page: `https://code.google.com/p/android-apktool/`
[5]Androguard project page: `https://code.google.com/p/androguard/`

code of the hypervisor that runs the *Virtual Machine* (VM) hosting the testing environment. The main advantages of this approach are:

- as the analysis runs underneath the virtual machine, even the most privileged attacks can be detected;
- a malware can very hardly disrupt the analysis, since it is performed externally.

Two frameworks following this approach have been presented last year: DroidScope [44] and CopperDroid [37].

DroidScope is an Android malware analysis platform built on top of QEMU. As previously said, the emulator has been instrumented in order to maintain intact the original Android environment and perform external dynamic analysis. More specifically, the authors modified the translation phase from Android code to *Tiny Code Generator* (TCG), an intermediate representation used in QEMU, in order to insert extra instructions that allows to perform system call tracing and memory map analysis. Then, to reconstruct the two semantic levels of Android, the Linux OS level and the Dalvik VM one, *Virtual Machine Introspection* (VMI) has been exploited. An overview of DroidScope architecture is shown in **Figure 2.9**. An interesting feature of this framework is that it exposes a set of APIs that allow the development of plugins to perform both fine and coarse-grained analyses (e.g., system call, single instruction tracing and taint tracking).

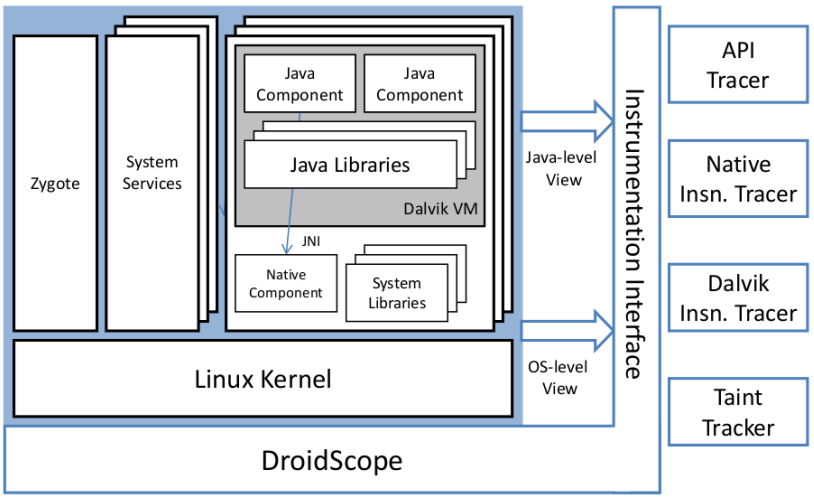

Figure 2.9: DroidScope architecture.

CopperDroid presents an approach very similar to DroidScope: as a matter of fact, it is a framework built on top of QEMU to automatically perform out-of-the-box dynamic behavioral analysis of Android malware. In particular, the authors instrumented the Android emulator in order to precisely

trace system call invocations at runtime. Afterwards, *Virtual Machine Introspection* (VMI) is exploited in order to analyze the sequence of collected system calls and extract behaviors both at low-level (OS-specific) and high-level (Android-specific). **Figure 2.10** presents an overview of CopperDroid architecture. The authors base their approach on the assumption that even high-level Android-specific behaviors are indeed achieved via system call invocations: as a matter of fact, they deeply inspected the structure of Binder, the Android-specific *Inter-Process Communication* (IPC) and *Remote Procedure Call* (RPC) mechanism, in order to develop a system call-centric analysis system. Furthermore, CopperDroid presents a new automatic stimulation approach based on the static information that can be retrieved from the manifest: the basic idea is to reach high code coverage triggering all the entry points (i.e., activities and services) of the application and to stimulate them generating a number of events of interest.

Finally, last year Google presented Google Bouncer [26], an automated scanning service developed to reject malicious applications from the official Google Play market. Little is known about it, except that it is a QEMU-based dynamic analysis framework. All the other information we have come from reverse-engineering attempts [35] that also showed how the Bouncer's analysis can be detected and thus evaded [21].

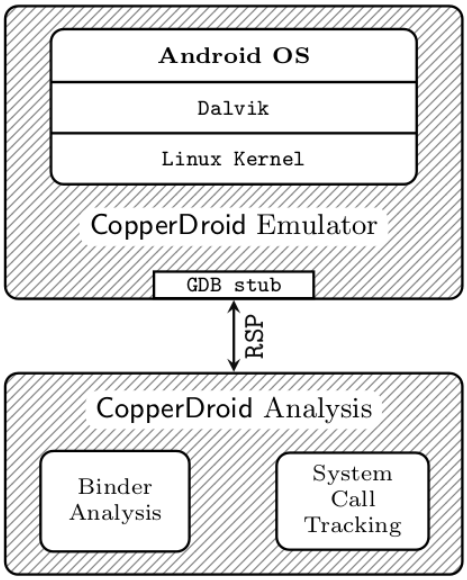

Figure 2.10: CopperDroid architecture.

2.2.2 Exercising of Android applications

In order to analyze the dynamic behavior of a program, it is of primary importance to correctly exercise it. In particular, the main goals of an effective exercising are two: reach the highest possible code coverage and generate a stimulation as similar as possible to the real case usage. Satisfying these requirements in modern touchscreen-based GUIs is not trivial: in this section we want to cite some existing approaches that face Android UI stimulation both for testing and dynamic analysis purposes.

One of the most used tools for automatic testing of Android applications is Monkey. This tool is included in the Android SDK and can generate pseudo-random UI events in order to stress-test an application. It is used in different automated dynamic malware analysis systems (e.g., Andrubis bases its automated stimulation approach on it). Its native integration in the Android system makes this tool very usable, however its pseudo-random event generation can unlikely lead to an efficient stimulation. We will describe Monkey in details it in *Section A.3*.

Another tool, always provided by the Android SDK, designed for functional-level testing of Android applications is MonkeyRunner. While Monkey directly runs inside the emulator/device, MonkeyRunner controls the emulator from a workstation by sending specific commands and events from an API. This tool can run automated tests on Android applications, but the developer has to provide the list of events that have to be executed.

An interesting work that proposes an improvement of the simple fuzz testing of Monkey is Dynodroid [28]. The basic idea is to apply an *"observe and execute"* approach (i.e., first of all analyze the contents of the window displayed and then generate random input events that can stimulate View elements able to consume them). In this way, Dynodroid is able to reach the same code coverage obtained with Monkey but generating much less events. In order to reach this result, the authors leverage MonkeyRunner and HierarchyViewer, another tool of Android SDK used to inspect the application UI (it will be described in detail in *Section A.4*).

The most interesting work on UI stimulation is surely SmartDroid [45]: this framework leverages static and dynamic analysis in order to extract a sequence of UI events that allow to stimulate sensitive behaviors, like sending SMS or getting user private data. The aim of the authors is to provide a tool to efficiently stimulate Android applications during dynamic malware analysis. The initial static analysis phase is characterized by three steps. First of all, SmartDroid builds the *Function Call Graph* (FCG) and the *Activity Call Graph* (ACG) of the application. Then bytecode instructions are extracted from the APK and analyzed in order to find sensitive method invocations.

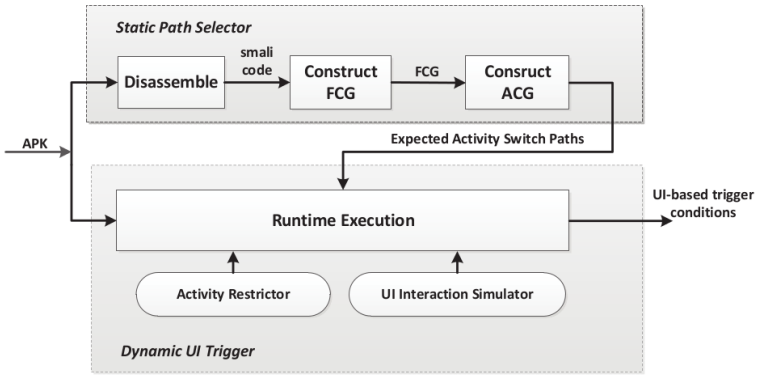

Figure 2.11: SmartDroid architecture and workflow

Finally, sensitive paths from application's entry points to sensitive method invocations are built. Thereafter, an automated dynamic analysis approach is used to verify the validity of the paths previously found. In order to do that, the authors modified the original Android code: in this way they can force the execution to try to follow the sensitive paths and verify if a sensitive behavior is actually triggered. **Figure 2.11** presents an overview of SmartDroid architecture and workflow.

Finally, we want to cite also RERAN [12], a tool for Android that allows to record and replay low-level UI events directly reading from and writing on system input devices. This work uses an approach very similar to the one used by PuppetDroid to inject input events, but it is limited to propose a mere re-execution of the original recorded touch events without offering any analysis of application *User Interface*: as we will see in *Section 3.3.7*, this re-execution approach is not very robust.

2.3 Open problems and Goals

As mentioned before, the main limitation of dynamic analysis is its inability to reach high code coverage: dynamic analysis can examine the actions performed in an execution path only if that path is actually explored. This is a particularly thorny problem for malware analysis: if the malware is not properly exercised there is the possibility that it does not expose its malicious behavior and consequently analysts end up to label it as good software. We previously described (*Section 2.2.1* and *2.2.2*) current solutions that try to mitigate this limitation integrating automatic exercising techniques in their frameworks. These techniques usually leverage stress-test tools, like Monkey, or a combination of static analysis and automatic *User Interface* (UI) exercisers, like SmartDroid. The problem of stress-test tools is that they rely

on pseudo-random generation of UI input events: randomly stimulating UI elements displayed on the screen can hardly reproduce the typical usage of human users, providing in this way low code coverage and unsatisfactory stimulation of interesting malicious behaviors. Exercising mobile applications in a proper way is not trivial problem, because these applications make use of highly-interactive UIs in order to leverage the capabilities of modern touchscreen devices, making then automatic exercising of their interfaces harder in respect to conventional PCs' scenario. For this reason, approaches like DynoDroid or SmartDroid leverage static analysis to reconstruct the semantic of UI elements on the screen in order to find execution paths that expose malicious behaviors. However, as we said, static analysis is ineffective in case of obfuscated code or dynamic payloads, two techniques widely used by modern malware: in presence of these two evasion techniques, automatic exercising approaches based on static analysis cannot properly identify malicious execution paths and, consequently, they cannot provide an effective UI stimulation.

In this work we then focus on the problem of application exercising for malware analysis purposes. We propose a new approach to exercise the UI of an Android application in order to change the way malware analysis experiments are currently conducted and effectively stimulate potentially malicious behaviors. To this end, we have to find a practical way to reproduce the typical usage of a potential victim of the malware. As said, automatic exercising approaches fail in reaching this intent for multiple reasons: our key intuition is to leverage human-driven exercising. As a matter of fact, human users can properly interact with the UI of an application because, unlike automatic exercisers, they understand the semantic of elements displayed on the screen and can exercise the application accordingly. In order to analyze the huge amount of application published every day, we cannot only leverage human-driven exercising. Analyzing security reports by AV vendors [7, 42], we discovered that a quite common practice among mobile malware authors is the repackaging of already existent malware samples in order to obtain slightly different variants able to avoid *Anti-Virus* detection techniques. We leverage this phenomenon to make our approach scale: in fact, our idea is to record a trace of the human-driven UI stimulation, that we name *stimulation trace*, performed during a test and leverage code similarity to automatically re-execute this trace on applications similar to one originally tested by the user. In this way, if at least one user in our system succeeds in manually stimulating a malicious behavior in a malware, it is quite likely that by reusing the same stimulation trace on similar applications, we can stimulate similar malicious behaviors.

The goal of this work is then to define and develop an Android sandboxed environment able to support both manual application testing, in order to collect new stimulation traces, and automatic application exercising, which

leverages previously recorded UI stimulation traces. Furthermore, we also want to experimentally verify if manual exercising allows to better stimulate malicious behaviors than automatic exercising techniques. Finally, we want to experimentally validate our approach and demonstrate that it brings an original contribution to the state of the art, allowing to overcome current limitations in application exercising for malware analysis purposes.

Chapter 3

PuppetDroid

In this chapter we describe PuppetDroid, a remote execution environment designed to exercise Android applications and perform dynamic malware analysis. As explained in *Section 2.3*, we designed PuppetDroid with two main goals in mind:

1. To provide a sandboxed environment that allows to safely perform manual tests on malicious applications and, at the same time, to record user interaction with the UI of the application.

2. To implement a *re-runner* framework in order to automatically exercise unknown applications, leveraging previously recorded stimulation traces.

To accomplish the first goal, we developed our framework in order to exploit remote sandboxex to safely run applications and, at the same time, to allow users to seamlessly interact with an application as if it was running locally on their device. Thanks to this solution, we let users test potentially malicious Android applications on their personal devices, avoiding any possible risk of infection or information leaking to them. As described in *Section 3.3.7*, we extended our sandboxed environment in order to store user interaction with the application in stimulation traces: with this term we indicate the sequence of actions performed by the user and the list of UI elements actually stimulated during the test. In addition to this, the sandbox has been properly instrumented to collect information about application behavior through dynamic analysis techniques.

To make PuppetDroid exercising approach scale, we designed our framework in order to support the re-use of stimulation traces, collected through manual exercising, on applications with a layout similar to the one originally tested. To this end, we developed Puppet ReRunner, our stimulation traces re-execution tool, whose description is presented in *Section 3.3.7*.

Thanks to these functionalities, our framework is able to collect efficient human-driven stimulation traces and re-use them on similar applications, making in this way our approach scale. In particular, PuppetDroid relies on crowd-sourcing to collect new stimulation traces: we believe that our system can attract the interest not only of security analysts but also of normal users that want to safely try potentially malicious applications they found on the web or in alternative markets. Finally, as last resort, we can leverage the abundance of crowd-sourcing services, like Amazon Mechanical Turk[1], in order to retrieve human workers to generate new stimulation traces.

In the following sections we present PuppetDroid system, starting with an introduction of the logical workflow of the approach (*Section 3.1*), presenting then an overview of its architecture(*Section 3.2*) and going on illustrating some implementation details that allow to understand how the whole system has been developed (*Section 3.3*).

3.1 Approach overview

In this section we present the typical usage scenario of PuppetDroid in order to illustrate the logical workflow of our approach. **Figure 3.1** shows the UML activity diagram of analyst interaction with our system.

The first step to be performed by the user is the selection of the APK to be tested: the user can upload a new APK to our system, choose one of the samples already available or retrieve an APK from Google Play.

In case the selected APK has been already tested, the user can examine the results of previously executed tests. These results illustrate the output of the dynamic analysis and indicate if the test has been performed leveraging manual exercising or re-executing an available stimulation trace. If the available test results do not satisfy the user, he or she can manually exercise the application: in this case the user leverages our Android application in order to directly interact with the sample. Upon test termination, the results are shown to the user that can decide if execute another manual test or terminate his or her activity.

If the selected APK has never been tested, our system looks for available stimulation traces associated to similar samples: if any, the user can decide to automatically exercise the application using one stimulation trace available. The rerun test is automatically performed by the system and, at the end of stimulation trace re-execution, test results are shown to the user. The user has then the possibility to perform another rerun test leveraging a different

[1] Amazon Mechanical Turk service page: `https://www.mturk.com/`

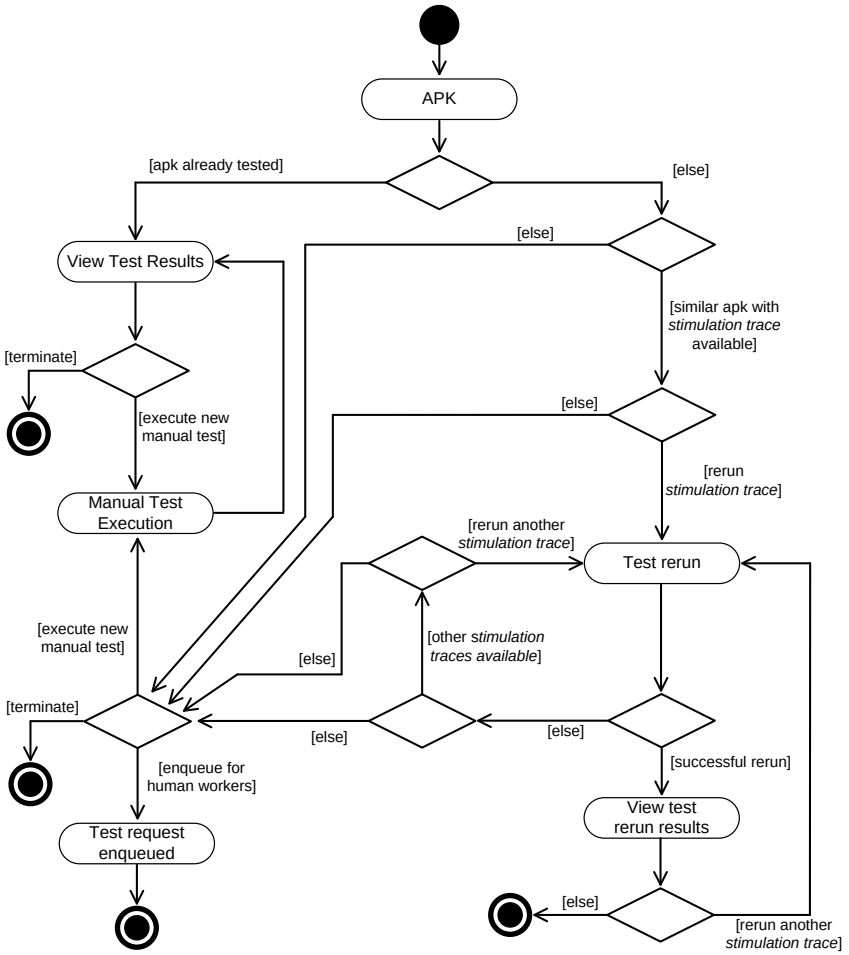

Figure 3.1: Activity diagram of user interaction with PuppetDroid system.

stimulation trace, if available, or to manually exercise the application or, finally, to terminate his or her activity.

In the diagram we also present the possibility to enqueue a test for human workers as alternative to manual exercising: as we previously mentioned, our system can leverage crowd-sourcing services, then this option indicates the possibility to ask our system to retrieve a turk (i.e., a human worker in the crowdsourcing terminology) to manually test the application.

This is the methodology we propose to test an unknown application. However, current implementation of PuppetDroid has been developed for testing purposes in order to validate our approach: for this reason it slightly differs from the approach we just presented. In particular, at the moment, the re-execution functionality is performed in a different way: the user does not choose an available stimulation trace in order to automatically exercise

an unknown application, but it choose one of the stimulation traces available on our systems and then PuppetDroid automatically rerun this trace on all the application similar to the one originally tested through manual exercising. This particular implementation helped us in the collection of experimental results: when PuppetDroid will be published, its implementation will be modified in order to fit the logical workflow we presented in this section.

3.2 System overview

In this section we provide a high-level introduction of PuppetDroid system, leaving the implementation details to the following sections.

Section 3.2.1 provides a high-level description of the system architecture, introducing the main components that constitute it.

Section 3.2.2 introduces the basic workflow of the system, from the moment the user uploads the application on PuppetDroid server, till the time the results are available.

3.2.1 System architecture

This section provides a high-level description of PuppetDroid system: each component presented here will be described in detail in the following sections.

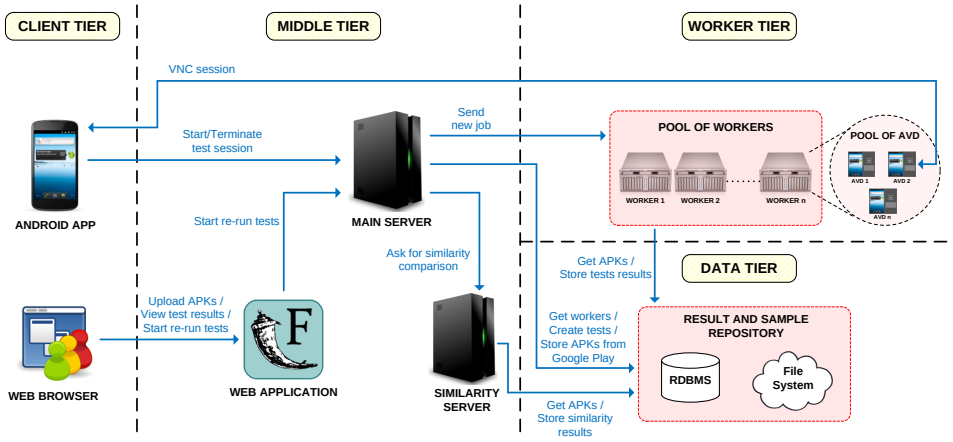

Figure 3.2: Architecture of PuppetDroid system.

PuppetDroid has a *multi-tier architecture* (**Figure 3.2**) to allow decoupling the different system components and facilitate future modifications and

improvements.

Client tier – It includes the two entry points used by the user to interact with the system: the web frontend, used to upload new APKs, view test results and start re-run tests, and the Android application, used to start and execute manual test sessions.

Middle tier – It is composed by the PuppetDroid core system: the *Main server*, which has the task of communicating with Android application, creating tests and finding available workers to perform them, the *Web Application*, which receives the samples uploaded by users and shows test results, and, finally, the *similarity server*, which has the task of calculating the similarity score between two applications.

Data tier – It includes a storing system for the results of the tests, the repository for the uploaded samples and the repository for the similarity comparisons.

Worker tier – This is the level where worker machines reside. Each worker is devoted to execute test requests received from the main server.

PuppetDroid is a highly interactive system, because when a new test session request is sent to our servers, we need a worker with an available sandbox to actually host that test session: to address this point, the *Worker Tier* has been designed to have a distributed architecture, so as to easily scale by simply increasing hardware resources.

3.2.2 PuppetDroid workflow

We present here the basic workflow of PuppetDroid, showing what happens behind the scenes when the user interacts with our system. The user can interact with PuppetDroid system in two ways: using the web application, to upload the APKs he/she wants to test, to see the results of tests previously executed tests and to leverage our re-execution functionality, or using the Android application, to actually manually exercising the application.

Figure 3.3 shows the basic workflow that let the user manually test an application directly provided by him. We can identify 8 main steps: the steps colored in green indicate actions that must be taken by the user, while blue ones indicate actions that are internally performed by the system.

1. **APK upload**: through our web frontend the user uploads the APK he/she wants to test.
2. **Sample storage**: after performing some consistency and security

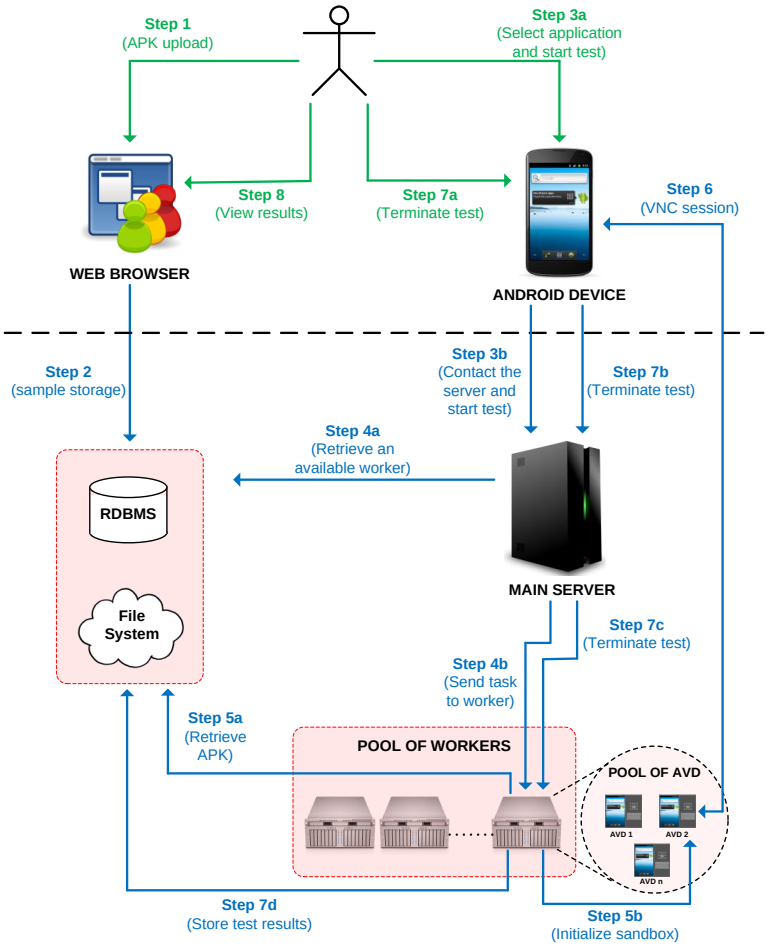

Figure 3.3: Workflow of the manual test of a sample provided by the user.

checks, the uploaded sample is stored into the PuppetDroid sample repository.

3. **Start test**: through our Android application the user selects the app he/she wants to test and start a new test session. The Android application contacts our main server and sends the request.

4. **Send task to worker**: our main server checks if there is an available worker and, if found, sends the task request to it.

5. **Sandbox initialization**: the selected worker retrieves the APK to be tested from the sample repository and initializes the sandbox that will hold the test session.

6. **VNC session**: when the sandbox is ready, a VNC (*Section 3.3.6*) channel is established with the Android application and the user can

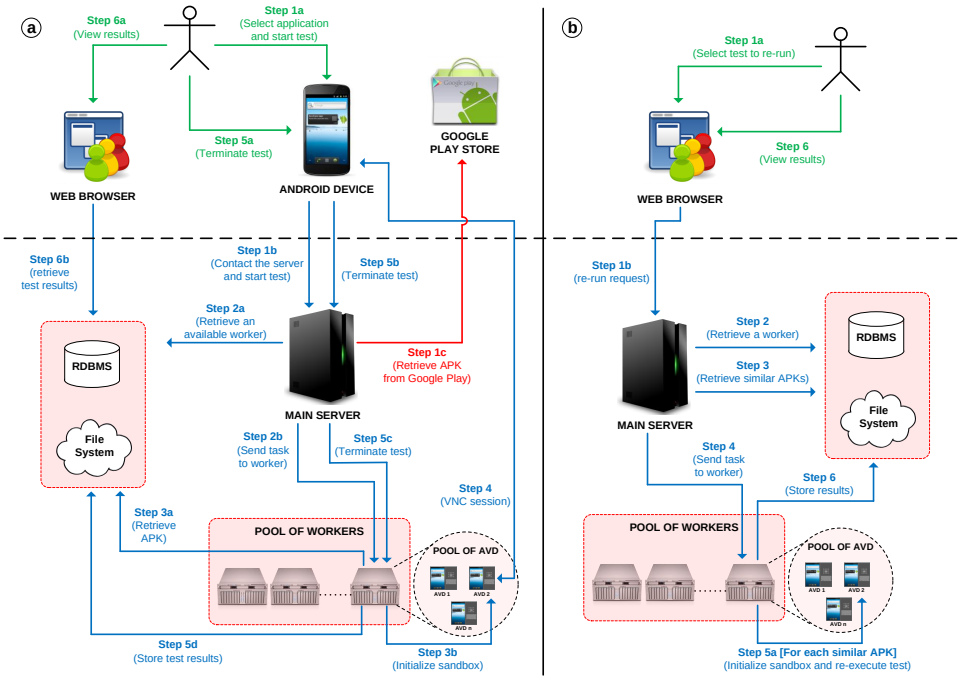

Figure 3.4: (a) Workflow of the manual test of a sample retrieved by Google Play. (b) Workflow of a test re-run.

interact with the sandbox.

7. **Test termination**: when the user decides to terminate the test, the Android application contacts our main server that sends a termination request to the worker server. The worker terminates the sandbox and stores the results on PuppetDroid result repository.

8. **Result view**: the user can now view test results using our web frontend.

This is the basic workflow to perform a manual test on PuppetDroid. Our system also allows to test free applications retrieved from Google Play, the official Android application market: for sake of completeness, we also show the workflow in this case in **Figure 3.4a**. As you can see, the only difference is that the user has not to previously upload an APK, but he/she chooses it using our Android application. The main server has the task of connecting to Google servers and retrieving the desired APK.

Finally, we show the workflow of an automatic test re-run in **Figure 3.4b**. As first step, the user has to select, through the web frontend, which test he/she wants to leverage to perform re-execution. Thereafter, our system retrieves the list of APKs similar to the one originally tested and automatically re-execute the stimulation trace associated to the chosen test on these

applications. We remind that current version of re-execution functionality has been implemented for testing purposes in order to validate our approach. In the final implementation, the user should choose the APK he/she wants to test and then select one of the available stimulation traces on our system to automatically exercise that application.

3.3 Implementation details

In this section we provide a detailed description of PuppetDroid main components. We start introducing the internal communication protocol used in our system (*Section 3.3.1*). We then present a description of the components that make up our multi-tier architecture: the Storage system (*Section 3.3.2*), the Main Server (*Section 3.3.3*), the Workers (*Section 3.3.4*) and finally the Web Application (*Section 3.3.5*). We conclude the chapter providing some technical details on the implementation of two fundamental aspects of our system: the VNC integration (*Section 3.3.6*) and the test re-execution (*Section 3.3.7*).

3.3.1 Communication protocol

This section describes the protocols used by the different components of PuppetDroid system to communicate with each other. Two protocols have been defined: the first one is used for the communication between the Android client application and the PuppetDroid main server, while the second is used for the internal communication between the main server and the worker servers.

3.3.1.1 Communication between clients and main server

This section describes the protocol used for the communication between the Android client application and the PuppetDroid main server. At the beginning of the communication, the server sends a string message to let the client know that it is speaking with the correct endpoint (**Table 3.1**). After this simple handshake phase, the client sends a byte indicating its request (**Table 3.2**). **Table 3.3** shows the values that *request-code* can take: 1 is used to ask for a new manual test session, 2 is used to terminate the current test session and save the results, 3 is used to abort the current test session and discard the results, while 4 is used to retrieve the list of available APKs.

N. of bytes	Type	Description
4	U32	*identification-string-length*
identification-string-length	U8 array	*identification-string*

Table 3.1: Identification message (from server).

N. of bytes	Type	Description
1	U8	*request-code*

Table 3.2: Request message (from client).

Start a new test session

In case of a new manual test session request, the client sends information about the device on which it is running (**Table 3.4**). In order to correctly initialize the emulator instance that will host the test, we need to know the Android *Operating System* (OS) version (*API-level*), the ABI and the information about screen size and DPI. Thereafter, if the client needs to test an application on Google Play, it sends Google authentication info to the server (**Table 3.5**). Finally the client has to specify which application has to be tested: first of all it sends a byte to indicate if the app has to be retrieved from Google Play or if it already present on PuppetDroid repository; then it sends a string indicating the package name of the app, in case it has to be downloaded from Google Play, or the ID used to identify the application on PuppetDroid system, if the application has been previously uploaded (**Table 3.6**).

This information is sent by the main server to an available worker server that properly initializes a sandbox for the test. When the initialization phase is completed, the server sends to the client a result message (**Table 3.8**). The first 4 bytes of server response indicate the result: if it is negative, it means that an error occurred during the initialization phase. In this case the server sends a byte indicating which type of error occurred (*error-code*). Otherwise, if the result is positive, it means that no error occurred and the result value indicates the port number that will be used for VNC communication: in this

Value	Description
0x01	*Connect request*
0x02	*Disconnect request*
0x03	*Abort request*
0x04	*APK list request*

Table 3.3: Possible values for *request-code*.

N. of bytes	Type	Description
1	U8	*API-level*
4	U32	*abi-string-length*
abi-string-length	U8	*abi-string*
4	U32	*screen-width*
4	U32	*screen-height*
4	U32	*screen-dpi*

Table 3.4: Device info messages (from client).

N. of bytes	Type	Description
1	U8	*auth-enabled*
4	U32	*android-id-length*
android-id-length	U8	*android-id*
4	U32	*auth-token-lenght*
auth-token-lenght	U8	*auth-token*

Table 3.5: Authentication messages (from client).

N. of bytes	Type	Description
1	U8	*app-source*
4	U32	*package-name-length*
package-name-length	U8	*package-name*
4	U32	*app-ID-length*
app-ID-length	U8	*app-ID*

Table 3.6: Package name messages (from client).

Value	Description
0x01	*Google Play*
0x02	*PuppetDroid Server*

Table 3.7: Possible values for *app-source*.

N. of bytes	Type	Description
4	U32	*result*
4	U32	*VNC-IP-length*
VNC-IP-length	U8	*VNC-IP*
4	U32	*test-ID-length*
test-ID-length	U8	*test-ID*
1	U8	*error-code*

Table 3.8: Result message (from server).

Value	Description
0x00	*AVD_ERROR*
0x01	*AUTH_ERROR*
0x02	*APK_ERROR*
0x03	*UPLOAD_TOKEN_ERROR*
0x04	*APK_SOURCE_ERROR*
0x05	*TEST_NOT_FOUND*
0x06	*APK_FILE_NOT_FOUND*
0x07	*NO_WORKER_AVAILABLE_ERROR*
0x08	*COMMUNICATION_ERROR*
0x09	*UNKNOWN_REQUEST_ERROR*
0x10	*UNKNOWN_ERROR*

Table 3.9: Possible values for *error-code*.

case the server can send the IP that the client has to use to start the VNC session and the ID used to uniquely identify the test session.

Close the current test session

When the client terminates the test session, it sends a disconnect request in order to inform the server to terminate the test and save the results. To correctly identify which test session has to be terminated, the client sends the IP and the port used for the VNC communication and the ID that identifies the test session (**Table 3.10**).

N. of bytes	Type	Description
4	U32	*VNC-IP-length*
VNC-IP-length	U8	*VNC-IP*
4	U32	*VNC-port*
4	U32	*test-ID-length*
test-ID-length	U8	*test-ID*

Table 3.10: Disconnect message (from client).

If the information provided matches a running test session, the server terminates the session, deallocates the resources associated to it and stores the results.

Abort the current test session

This request is handled in a similar way the close session request is managed: the only difference is that the server does not store test results but it discards them.

Retrieve the APK list

Before starting a new test session, the client has to know which applications can be actually used for the test. This request allows retrieving the list of available APKs from the PuppetDroid server.

First of all the client sends to the server the string to be used in the search; then it indicates where the application should be retrieved, if from Google Play or from PuppetDroid APK repository (**Table 3.11**).

N. of bytes	Type	Description
4	U32	*search-string-length*
search-string-length	U8	*search-string*
1	U8	*app-source*

Table 3.11: Get APK list message (from client).

app-source can take the same values shown in **Table 3.7**. In case the client asks for an application from Google Play, it has to send Google authentication info to the server (**Table 3.12**).

N. of bytes	Type	Description
4	U32	*android-id-length*
android-id-length	U8	*android-id*
4	U32	*auth-token-lenght*
auth-token-lenght	U8	*auth-token*

Table 3.12: Authentication messages (from client).

The server uses the info received to retrieve the list of APKs matching the search request then it sends the number of APKs found to the client (**Table 3.13**). For each application found, the server sends to the client the application and package names, the version, the APK ID used to identify the sample on PuppetDroid repository, if the source is PuppetDroid server, and finally the app icon image (**Table 3.14**).

N. of bytes	Type	Description
1	U8	*num-apks-found*

Table 3.13: APK list length message (from server).

N. of bytes	Type	Description
4	U32	*app-name-length*
app-name-length	U8	*app-name*
4	U32	*package-name-lenght*
package-name-lenght	U8	*package-name*
4	U32	*version-length*
version-length	U8	*version*
4	U32	*apk-ID-lenght*
apk-ID-lenght	U8	*apk-ID*
4	U32	*icon-image-size*
icon-image-size	U8	*icon-image*

Table 3.14: APK info message (from server).

3.3.1.2 Communication between workers and main server

This section describes the protocol used for the communication between the PuppetDroid main server and the worker servers. The communication is always started by the main server and it is used to start a new test session, to terminate or abort a currently running test or to ask the worker to automatically exercise an application re-running a previously recorded stimulation trace (**Table 3.15**).

N. of bytes	Type	Description
1	U8	*request-code*

Table 3.15: Request message (from main server).

Start a new test session

When the server receive a connect request from a client, it looks for an available worker to start a new test session. In order to correctly initialize the emulator on which the test will be executed, the main server sends to the worker the info retrieved from the client and the info needed to retrieve the APK that has to be installed on the emulator (**Table 3.17**).

The worker uses this information to start a new emulator for the test session. When it terminates the initialization phase, it sends a result message: if positive, its value indicates the worker port used for VNC session, else, if

Value	Description
0x01	*New test request*
0x02	*Terminate request*
0x03	*Abort request*
0x05	*Re-run request*

Table 3.16: Possible values for *request-code*.

negative, it signals that something went wrong and a further byte is sent to indicate the type of error occurred (**Table 3.18**).

N. of bytes	Type	Description
1	U8	*API-level*
4	U32	*abi-string-length*
abi-string-length	U8	*abi-string*
4	U32	*screen-size-string-length*
screen-size-string-length	U8	*screen-size-string*
4	U32	*screen-dpi*
4	U32	*package-name-length*
package-name-length	U8	*package-name*
4	U32	*apk-url-length*
apk-url-length	U8	*apk-url*
4	U32	*test-url-length*
test-url-length	U8	*test-url*

Table 3.17: Device info messages (from main server).

N. of bytes	Type	Description
4	U32	*result*
1	U8	*error-code*

Table 3.18: Result message (from worker server).

Terminate a test session

To terminate a running test session, the main server has to sent the ID that uniquely identifies the test(**Table 3.19**). The worker terminates the emulator associated to the test and stores the results.

Abort a test session

As for the terminate request, the worker only needs the test ID to abort a running test session (**Table 3.19**).

N. of bytes	Type	Description
4	U32	*test-ID-length*
test-ID-length	U8	*test-ID*

Table 3.19: Terminate test message (from main server).

Re-run a test

In case of a re-run request, the worker needs the following information:

- *Emulator settings* – to correctly initialize the sandbox that will host the test.
- *Test ID* – to retrieve stimulation trace files needed to the re-runner.
- *APK list* – the list of similar APKs on which the test should be re-executed

N. of bytes	Type	Description
1	U8	*API-level*
4	U32	*abi-string-length*
abi-string-length	U8	*abi-string*
4	U32	*screen-size-string-length*
screen-size-string-length	U8	*screen-size-string*
4	U32	*screen-dpi*
4	U32	*test-ID-length*
test-ID-length	U8	*test-ID*
4	U32	*num-apks*

Table 3.20: Re-run info messages (from main server).

num-apks indicates the size of the APK list: for each APK the main server sends the URL needed to retrieve it from PuppetDroid sample repository (**Table 3.21**).

N. of bytes	Type	Description
4	U32	*apk-url-length*
apk-url-length	U8	*apk-url*

Table 3.21: Re-run info messages (from main server).

When the worker terminates to re-execute the stimulation trace associated to the selected test on all applications in the list, it sends a result message to the main server: if result value is negative, it means an error occurred and a further byte is sent in order to indicate the type of error. If no error occurs during the re-execution, the result has value 0 (**Table 3.22**).

N. of bytes	Type	Description
4	U32	*result*
1	U8	*error-code*

Table 3.22: Result message (from worker server).

In the final implementation of PuppetDroid, the worker will not receive a list of applications on which perform re-run tests but only the application the user selected to be automatically exercised.

3.3.2 Storage

PuppetDroid storage system is built on top of 2 main components: a *Relational Database Management System* (RDBMS) and a shared filesystem layer.

PuppetDroid leverages an *Object Relational Mapper* (ORM) to access its RDBMS (specifically, it uses SQLAlchemy mapping over MySQL). This ensures high decoupling between data models and the chosen database system, in addition to some other features provided out of the box by the specific ORM (such as session and transaction management, lazy data loading and SQL injection mitigation). The *Entity-Relationship* (ER) diagram of the PuppetDroid database schema is reported in **Figure 3.5**.

As we previously mentioned, PuppetDroid stores the metadata of any newly uploaded sample in the database. The table responsible for keeping this information is named `apk_sample`. It includes the following attributes:

`md5sum` – the sample MD5 hash value;

`sha1sum` – the sample SHA1 hash value;

`sha256sum` – the sample SHA256 hash value;

`package_name` – the Android package name;

`version` – the Android application version;

`app_name` – the Android application name;

`main_activity` – the Android application main activity;

`apk_url` – the sample file path;

`icon_url` – the sample image icon path;

`source` – indicates if the file has been uploaded through the web applica-

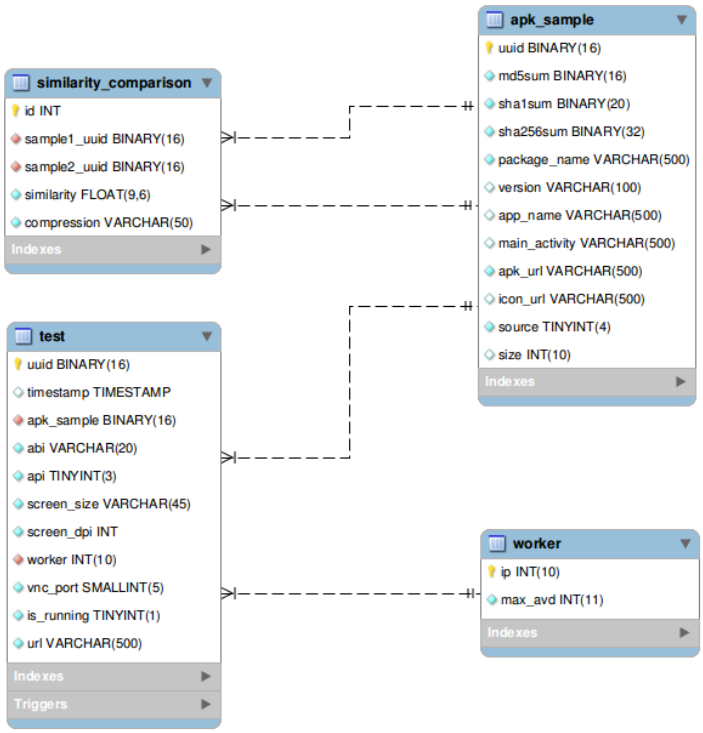

Figure 3.5: PuppetDroid database *Entity-Relationship* diagram.

tion or it has been retrieved from Google Play;

`size` – the sample file size.

The database is also used to keep information about PuppetDroid worker servers: when a new worker is connected to system, a new entry is added to the DB in order to allow the *Main Server* to properly send jobs to it. The table responsible for keeping workers metadata is `worker`:

`ip` – the IP address of the worker server;

`max_avd` – the maximum number of concurrent sandboxes that can be hosted by the worker.

All metadata about test execution are stored in `test` table:

`timestamp` – a timestamp indicating when the test started;

`apk_sample` – reference to the APK sample tested;

`abi` – ABI of the user's device;

`api` – Android API level of the user's device;

`screen_size` – screen dimensions of the user's device;

`screen_dpi` – screen DPI of the user's device;

`worker` – IP address of the worker hosting the test;

`vnc_port` – port used in the VNC session;

`is_running` – flag indicating if the test is still in execution;

`url` – path to the folder containing test results.

Finally, we have `similarity_comparison` table to store information about the similarities between the samples in our repository:

`sample1_uuid` – reference to the first APK sample of the comparison;

`sample2_uuid` – reference to the second APK sample of the comparison;

`similarity` – similarity score between the two samples (expressed as percentage);

`compression` – compressor used during the comparison.

The concrete sample and test results storage is instead performed directly on a specific partition of the filesystem. The *Main Server* and the *Workers* can concurrently access to this partition using *SSH Filesystem* (SSHFS): this allows them to interface with a FUSE-based filesystem, without worrying about the specific filesystem implementation on each machine.

In particular, for each sample stored in our APK repository, we have a folder, identified by its UUID, containing the APK file, the image icon of the application and a text file containing some static information extracted from the manifest. The results for each test executed on our system are stored in a folder, identified by test UUID, containing the report files produced by the dynamic analysis sandbox (see *Section 3.3.4*), the files collecting information about UI stimulation trace (see *Section 3.3.7.1*) and a folder containing the results of the re-run tests. Each re-run test has a folder, identified by the UUID of the APK sample on which the stimulation trace has been re-executed, that contains the report files produced by the dynamic analysis sandbox and a log file produced by our ReRunner tool (see *Section 3.3.7*).

The final version of PuppetDroid will store re-run test in a different the way: each re-run test will be saved as a normal test and a column indicating the type of test, manual or re-run, will be added to `test` table.

3.3.3 Main Server

The *Main Server* has different system management roles:

- Management of requests from Android application clients.
- Creation of new tests and job delivering to workers.
- Balancing of the workload on available workers.
- Database management.
- Google Play APKs retrieval.

The communication with Android clients is managed through the use of TCP sockets: each request from the clients is processed in a different thread in order to ensure concurrency. To protect sensible data received from the client during the communication, as Google authentication data or device information, the socket channel is encrypted using TLS. Since is not possible to expect that each client has its own certificate, the authentication is only server-side. The current version of PuppetDroid uses self-signed certificates, so in order to allow the Android application to correctly connect to our server, the public certificate of the main server has been stored inside the APK, leveraging the Key and Certificate management tool `keytool` and the *Java Cryptographic Extension* (JCE) provider from Bouncy Castle.

The communication with the clients follows the protocol directions presented in *Section 3.3.1*.

When a new request is received by the *Main Server*, it has the task to dispatch the job to an available worker, trying to equally distribute the workload on the different workers. The communication with the worker servers is managed through the use of TCP sockets encrypted using TLS: in this case the authentication is bilateral and each worker is provided with a different certificate.

As previously mentioned, our system allows to test also APKs from Google Play: to retrieve the list of APKs on the official Android store and to actually download the desired APK, we leverage the *Google Play Unofficial Python API* developed by E. Girault [2]: this Python library simulates the network activity generated by the Android Google Play client application in order to communicate with Google servers and retrieve the desired information. The communication between Google Play servers and Android client application is based on Google's Protocol Buffers, a library that allows to encode messages in binary, compact blobs before sending them on the network. In order to correctly decode these messages, you have to know the structure of the exchanged messages: the author of this interesting library exploited Androguard[3] in order to reverse the official Android Google Play application

[2]Google Play Unofficial Python Play are released under the BSD License. The reader may refer to `https://github.com/egirault/googleplay-api` to retrieve the source code of the project.

[3]Androguard project page: `https://code.google.com/p/androguard/`

and retrieve the structure of the messages exchanged with Google servers. A detailed description on how this library has been developed can be found here [11].

In order to simulate the communication of the Android Google Play application, we need user's credentials to log in on Google Play servers: in particular, the PuppetDroid application retrieves from the user's device the `androidId`, the Google account name and a temporary GoogleAuthToken. These information are safely sent to our server using a TLS socket and they are only used for authentication purposes and discarded after the retrieval of the desired APK.

3.3.4 Workers

PuppetDroid tests are safely hosted by remote Android sandboxes transparently accessible to the user through our Android application: the *Worker Servers* have the task of hosting one or more of these sandboxes, managing the communication between the sandbox and the Android application and storing the results once a test is terminated.

As mentioned in *Section 3.3.3*, the *Worker* receives job requests from the *Main Server* through a TLS communication socket. The protocol adopted for the communication is shown in *Section 3.3.1*.

The core component of *Worker* implementation is the Android sandbox hosting the test session: it is basically an Android emulator instance, properly modified in order to perform dynamic malware analysis. As mentioned in *Section 2.2.1*, different Android sandboxes for dynamic malware analysis have been implemented so far and most likely others will be implemented in the future. In order to avoid to tie our system to a specific sandbox implementation, we implemented a Python class, named `avd`, to manage generic AVD instances: when we want to add the support for a new sandbox implementation, we only have to define a new Python class that inherits from the original `avd` class and to properly override methods related to dynamic analysis. **Figure 3.6** shows the life cycle common to all the sandboxes. The red colored boxes indicate the phases that are usually customized to meet the requirements of the specific sandbox implementations.

The first phase is the preparation of the environment to host the sandbox: during this phase the *Worker* receives information about sandbox features from the *Main Server* and looks for 4 free ports to be dedicate to the sandbox. In particular a couple of ports are requested by the ADB server to correctly communicate with the ADB daemon and a couple of ports are needed for the VNC session. Then the AVD instance is created using the command

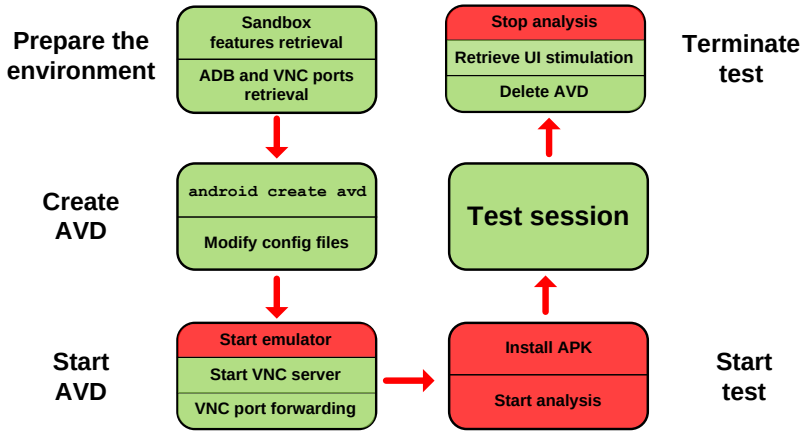

Figure 3.6: Sandbox life cycle diagram.

`android create avd`, as explained in *Section A.1*: to exactly adhere to user's device features, further changes are applied to AVD configuration files. The *Worker* can now start the emulator: this phase can be customized in order to properly load system images during emulator startup. During this phase the VNC server daemon is started and connected to the ports retrieved during the preparation phase. When the emulator is ready, it is initialized to host the test: this phase is usually customized according to the requirements of each specific sandbox. When the user terminates the test, the *Worker* stops the analysis, retrieves from the emulator the files storing UI stimulation data and delete the AVD. In case of a re-run test, the life cycle is quite the same, with the only differences that VNC operations are removed and the test session is automatically managed by our ReRunner tool.

We have successfully added to PuppetDroid the support for DroidBox and CopperDroid sandboxes following this approach. The current implementation of our system uses CopperDroid sandbox for multiple reasons: it has better performance, it allows to test more applications and it is currently still developed and supported. In the following section we provide some details about CopperDroid integration in our system.

3.3.4.1 CopperDroid

CopperDroid has been introduced in *Section 2.2.1*: it leverages an instrumented Android emulator to perform system call tracking and out-of-the-box system call-centric analysis. This tool has been realized by Royal Holloway University of London in collaboration with Università degli Studi di Milano: we had the possibility to cooperate with CopperDroid's authors in order to integrate their sandbox in our system.

Behavior	Blob	Hit
write	{'filename': u'/sys/qemu_trace/process_name'}	2
write	{'filename': u'/sys/qemu_trace/process_name'}	2
write	{'filename': u'/sys/qemu_trace/process_name'}	2
write	[MKDIR] /data/data/com.keji.danti80/files	1
write	{'filename': u'/data/data/com.keji.danti80/files/xxx.apk'}	1
write	{'filename': u'/sys/qemu_trace/process_name'}	2
write	{'filename': u'/sys/qemu_trace/process_name'}	2
write	[MKDIR] /data/data/com.sec.android.bridge/shared_prefs	1
write	[UNLINK] /data/data/com.keji.danti80/files/xxx.apk	1
connect	{'host': '10.0.2.3', 'retval': 0, 'port': 53}	1
outgoing_dns_query	{'query_data': 'b3.8866.org. 1 1'}	1
connect	{'host': '221.5.133.18', 'retval': -115, 'port': 8080}	1
write	{'filename': u'/data/data/com.sec.android.bridge/shared_prefs/first_app_perferences.xml'}	3
write	[UNLINK] /data/data/com.sec.android.bridge/shared_prefs/first_app_perferences.xml.bak	2
write	{'filename': u'/data/data/com.keji.danti80/files/atemp.jpg'}	1
write	[UNLINK] /data/data/com.keji.danti80/files/atemp.jpg	1
write	{'filename': '221.5.133.18 port:8080'}	2

Table 3.23: Example of behavior list generated by CopperDroid.

CopperDroid instrumented the QEMU hypervisor in order to intercept when the `swi` and the `cpsr_write` instructions are executed: `swi` is the ARM ISA instruction used for the invocation of system calls, triggering the software interrupt that causes the user-to-kernel transition, while `cpsr_write` is the instruction used, as the name suggest, to write on `cpsr` register. Monitoring the value contained in the `cpsr` register allows to keep trace if a switch from supervisor to user mode happened: this is a robust way to know if a system call terminated. After having established this approach to track system call invocations, the authors studied the structure of Binder, the Android-specific *Inter-Process Communication* (IPC) and *Remote Procedure Call* (RPC) mechanism, in order to extract human-readable information from the system calls they collect through CopperDroid. As final step, CopperDroid automatically analyzes the list of collected system calls in order to find semantic associations between them and therefore identify a list of behaviors performed by the analyzed application. The term behavior is used here to indicate a sequence of system call invocations identifying a high-level action, such as write on file or contact a remote server.

As result of the analysis, CopperDroid generates the following files:

`copper_trace.log` – a log file containing static information about the sample and the list of actions performed by CopperDroid framework;

`copper_trace.pcap` – a file containing the network traffic produced by the application;

`copper_trace.pickle` – a file containing the serialized Python data structures used to store information on recorded behaviors.

Table 3.23 shows an example of the behavior list extracted from the pickle file (`copper_trace.pickle`) generated by CopperDroid.

As said, CopperDroid only modifies QEMU, while it leaves unchanged Android system images loaded on emulator startup. The integration of this tool in PuppetDroid has been quite simple: it has been enough to override some methods of `avd` class in order to inform the system to use CopperDroid instance of QEMU and to store the behavior list files generated after test execution.

3.3.5 Web application

PuppetDroid web application is built on top of the Flask framework and let the user perform the following actions:

- Upload new APK on PuppetDroid system.
- View the list of available APKs.
- Examine test results.
- Ask for leverage an available stimulation trace in order to perform re-run tests.

The current version of our web application has been developed for testing purposes: at the moment there are not distinctions on the type of user accessing to it, so a generic user can access to the test results performed by others or test the APK samples uploaded by other users. We are planning to improve the current implementation of web application in order to provide user authentication and a better presentation of test results.

3.3.6 VNC implementation

As mentioned before, PuppetDroid leverages *Virtual Network Computing* (VNC) to allow the user to interact with the remote sandbox through our Android application. We decided to use VNC because it is a platform-independent de facto standard for graphical desktop sharing systems and

several open-source solutions have been implemented in the last years. VNC was originally developed as an internal solution at the UK Olivetti & Oracle Research Lab (ORL): when AT&T Research Lab acquired and subsequently closed the ORL, the original developers released the source code under the *GNU General Public License* (GNU GPL).

Section 3.3.6.1 presents the protocol on which VNC is based on. *Section 3.3.6.2* and *3.3.6.3* show how we exploited existent open-source VNC solutions in order to integrate it in our system.

3.3.6.1 Protocol description

Virtual Network Computing (VNC) has a client/server architecture, where the *server* is the endpoint that actually shares its screen and let the clients take control of it and the *clients*, or *viewers*, watch, interact and eventually control the server (**Figure 3.7**).

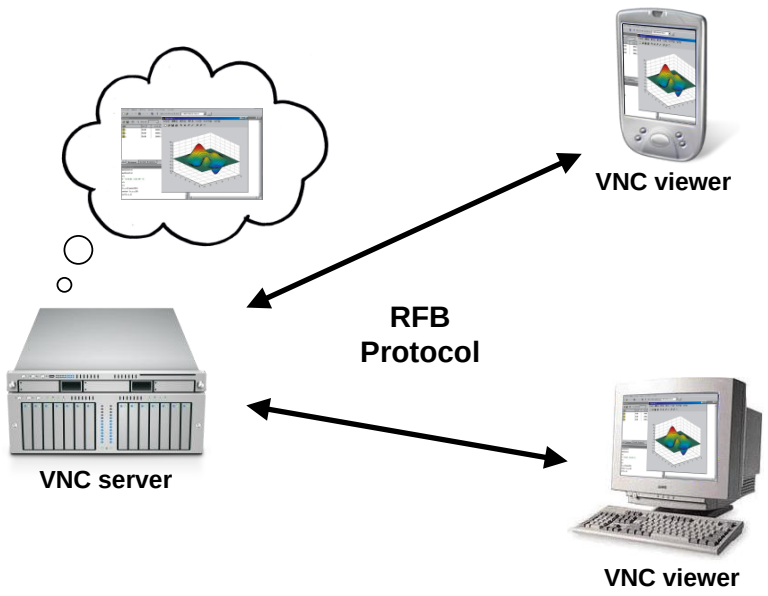

Figure 3.7: *Virtual Network Computing* (VNC) client/server architecture.

VNC is based on *Remote Framebuffer* (RFB), a simple protocol for remote access to *Graphical User Interfaces*. Because it works at the framebuffer level, it is applicable to all windowing systems and applications: the protocol has been designed to make the clients as simple as possible in order they could run on the widest range of hardware. The protocol also makes the client stateless: if a client disconnects from a given server and subsequently reconnects to that same server, the state of the user interface is preserved.

The display side of the protocol is based around a single graphics primitive: "*put a rectangle of pixel data at a given x,y position*". As the same authors state, it might seem an inefficient way of drawing many user interface components: however, allowing various different encodings for the pixel data gives a large degree of flexibility in how to trade off various parameters such as network bandwidth, client drawing speed and server processing speed. A sequence of these rectangles makes a *framebuffer update*, where an update represents a change from one valid framebuffer state to another. The update protocol is demand-driven by the client: an update is only sent from the server to the client in response to an explicit request from the client. This gives the protocol an adaptive quality: the slower the client and the network are, the lower the rate of updates becomes. With a slow client and/or network, transient states of the framebuffer can be ignored, resulting in less network traffic and less drawing for the client.

The input side of the protocol is based on a standard workstation model of a keyboard and multi-button pointing device, but it can be easily adapted to any I/O device, such the modern touchscreen systems. Input events are simply sent to the server by the client whenever the user presses a key, touches the screen or the pointing device is moved.

Initial interaction between the RFB client and server involves a negotiation of the *format* and *encoding* that will be used to send the pixels. As previously said, the protocol has been designed to make the client as simple as possible: the basic idea is that the server must always be able to supply pixel data in the form the client wants. However if the client is able to cope equally with several different formats or encodings, it may choose one that is easier for the server to produce. Pixel *format* refers to the representation of individual colors by pixel values. The most common pixel formats are 24-bit or 16-bit "true color" and 8-bit "color map". *Encoding* refers to how a rectangle of pixel data will be sent on the wire. Every rectangle of pixel data is prefixed by a header giving the X,Y position of the rectangle on the screen, the width and height of the rectangle, and an encoding type that specifies the encoding of the pixel data. The data itself then follows using the specified encoding.

The currently main supported encodings are:

Raw – the simplest encoding type, supported by all client and server VNC implementations;

CopyRect – used to copy a rectangle of pixel data the client already has elsewhere in the framebuffer, avoiding to re-send it;

RRE – *rise-and-run-length encoding*, in which a rectangle of pixel data is partitioned into rectangular subregions, each of which consists of pixels of a single value, and the union of which comprises the original rectangular

region;

Hextile – a variation of *RRE*, where the rectangles are split up into 16x16 *tiles*;

ZRLE – stands for *Zlib Run-Length Encoding*, and combines Zlib compression, tiling, palettisation and run-length encoding;

Tight – uses a combination of Zlib and JPEG compression in order to maximize compression ratio and minimize CPU usage;

Ultra – uses LZO compression in order to compress and de-compress pixel data in real-time, favoring speed over compression ratio.

The communication between client and server can be divided in three stages: a first handshaking phase, the purpose of which is to agree upon the protocol version and the type of security to be used, an initialization phase, where the client and server exchange *ClientInit* and *ServerInit* messages, and a final stage dedicated to the normal protocol interaction. A detailed description of the RFB communication protocol can be retrieved from RealVNC official documentation [38].

3.3.6.2 VNC server

The first step in the integration of the VNC protocol in our system has been the development of an Android VNC server. There are a few open-source projects that take a stab in this challenge: the most interesting one is Fastdroid VNC[4], developed starting from the original framebuffer VNC server for the iPAQ and Zaurus and adapted to the Android system. Our initial VNC server implementation starts from this project, which we then modified and extended to satisfy our requirements. The project is based on LibVNCServer, a set of widely used, cross-platform C libraries that allow to quite easily implement VNC server functionality. These libraries provide all the low-level stuff needed to the development of a VNC server: encodings, RFB communication protocol, communication sockets and so on. The developer has only to implement "high-level" methods that allow to initialize server structure for the management of the framebuffer and to instruct the server about how to manage input events.

Android relies on the standard Linux frame buffer device, usually accessible at `/dev/graphics/fb0`. In particular, every displayed window gets implemented with an underlying *Surface* object, an object that gets placed on the framebuffer by *SurfaceFlinger*, the system-wide screen composer. Each *Surface* is double-buffered: the back buffer is where drawing takes place and the front buffer is used for composition. Android flips the front and back

[4]Fastdroid VNC server project page: `https://code.google.com/p/fastdroid-vnc/`

buffers, ensuring a minimal amount of buffer copying and that there is always a buffer for *SurfaceFlinger* to use for composition. In order to avoid frame misses during VNC session, it is fundamental to take into account the double buffering mechanism used by Android. In particular, to know which buffer has been last used by *SurfaceFlinger*, it is possible to call `ioctl` passing as request parameter `FBIOPUT_VSCREENINFO`: since the double buffer can be seen as a virtual display that has twice the size of the physical screen, the `yoffset` attribute of the `screeninfo` structure, obtained as result of the call, will say which buffer you have to consider.

Then, we can summarize the actions performed by the VNC server when it receives a *framebuffer update* in the following points:

- It uses `ioctl` to identify which buffer has to be checked.
- It compares the system framebuffer with the local copy of the framebuffer (updated at the last *framebuffer update* request).
- In case of discrepancies, it uses LibVNCServer libraries to notify the client.

The management of input events is quite simple: since we are working only on the Android emulator, the devices associated to the touchscreen and to the keyboard are fixed and respectively are `/dev/input/event1` and `/dev/input/event2`. Furthermore, since our VNC server implementation has to communicate only with our Android application, we established a convenient way to distinguish different touch events:

- Android touch up event is sent as a *PointerEvent* with *button-mask* equal to 0.
- Android touch down event is sent as a *PointerEvent* with *button-mask* equal to 1.
- Android touch move event is sent as a *PointerEvent* with *button-mask* equal to 2.

Finally, our VNC server implementation has been extended in order to allow UI stimulation recording, as better explained in *Section 3.3.7.1*.

3.3.6.3 VNC client

Our Android client application integrates in its implementation a VNC viewer. Since a lot of Java open-source VNC client implementations can be found on the web, we decided to exploit one of this solutions to realize our Android VNC client.

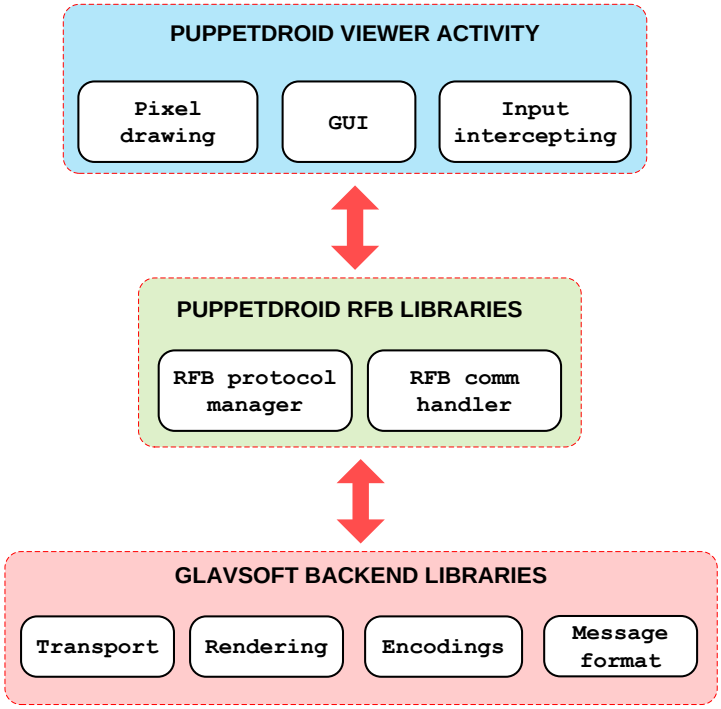

Figure 3.8: TightVNC client integration in PuppetDroid.

Our choice fell on TigthVNC Java Viewer Version 2[5], developed by GlavSoft LLC. The reasons that lead us to choose TightVNC are the following:

- The authors release the code under GNU GPL license.
- It is implemented in Java, then it can be easily integrated in Android.
- It is nearly as fast as the native version, always released by GlavSoft LLC.

As shown in **Figure 3.8**, of the original source code, we kept unchanged the backend used for the low-level management of the RFB protocol: encode/decode functions, message format structures and portions of the transport and rendering methods. Then we developed custom RFB and communication handlers, in order to interface with original TightVNC code, that are used by our Android *Viewer Activity* to display the results of the communication with the VNC server.

[5]TightVNC official website: `http://www.tightvnc.com/`

3.3.7 Puppet ReRunner

In this section we present how we developed our stimulation traces re-execution strategy. We remind that our re-execution approach bases on the assumption that if we succeed in stimulating a malicious behavior in a malware, it is quite likely that if we re-use the same stimulation trace to exercise an application similar to that malware, we are able to stimulate similar malicious behaviors. The motivations behind this assumption have been illustrated in *Section 2.3*.

First of all, we describe here the development and implementation of the technique we use to record user interaction with the application during a manual test: this technique allows to keep trace of the sequence of user actions and UI elements stimulated during the test and to save this information in stimulation traces. We then introduce Puppet ReRunner, the tool we developed to actually re-execute previously recorded stimulation traces on similar applications (*Section 3.3.7.1*). Last, we describe the solution adopted to calculate if two samples can be classified as similar (*Section 3.3.7.2*).

3.3.7.1 Input events recording

In this section we present the strategy used in PuppetDroid to record UI stimulation.

First of all, we need to keep trace of the sequence of input events generated by the human user. This can be done with the minimum effort thanks to how we structured manual test execution: as a matter of fact, to let the Android application interact with the remote sandbox we leverage VNC. This means that our system already receives the low-level input events generated by the user: our Android application intercepts these events and translates them in a sequence of RFB *PointerEvent* or *KeyEvent* messages that are sent to the VNC server. Therefore we extended our VNC server implementation to save on file the sequence of input events received. **Figure 3.9** shows the format used to store input events on file. For each event the following information is saved:

`timestamp` – the moment in which the event is executed;

`event_type` – the type of event, 0 for touch events and 1 for key ones;

`action` – the type of action performed, 0 for up, 1 for down and 2 for move;

`x_pos` – x position on the screen;

`y_pos` – y position on the screen;

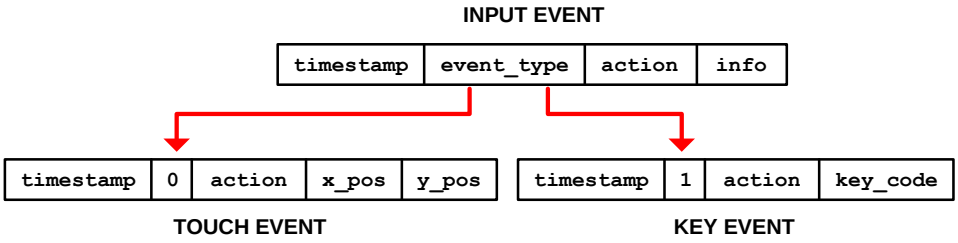

Figure 3.9: Format used to store input events.

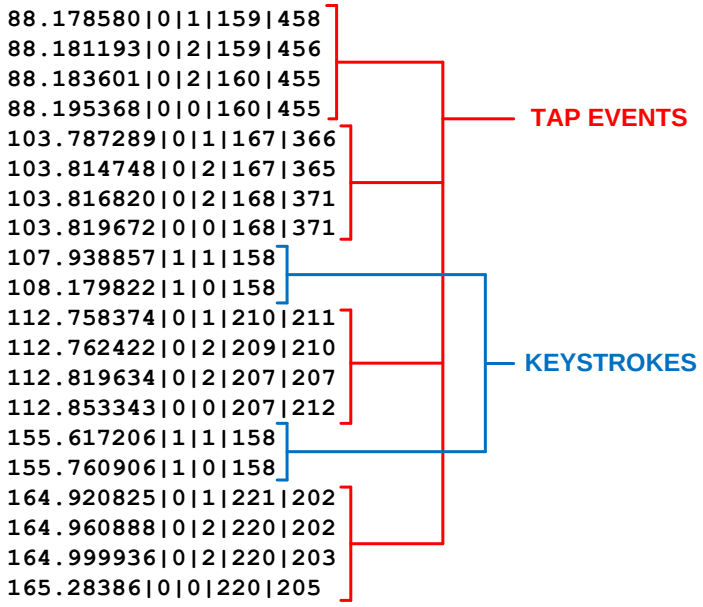

Figure 3.10: Excerpt of a sample input events file.

`key_code` – code used to identify which button has been pressed.

Figure 3.10 shows an excerpt of a sample input events file generated by the VNC server.

The input events file generated can be easily translated into a sequence of monkey events, as shown in **Listing 3.1** and **3.2**, in order to reproduce exactly the same sequence of input events of the original test. Unfortunately this approach is not reliable. As a matter of fact, it may happens that two similar application have about the same graphical layout but still there are some small differences, such as a button slightly shifted, that can make test re-execution fail.

Listing 3.1: Recorded input events

```
88.178580|0|1|159|458
88.181193|0|2|159|456
88.183601|0|2|160|455
88.195368|0|0|160|455
103.787289|0|1|167|366
103.814748|0|2|167|365
103.816820|0|2|168|371
103.819672|0|0|168|371
107.938857|1|1|158
108.179822|1|0|158
112.758374|0|1|210|211
112.762422|0|2|209|210
112.819634|0|2|207|207
112.853343|0|0|207|212
155.617206|1|1|158
155.760906|1|0|158
164.920825|0|1|221|202
164.960888|0|2|220|202
164.999936|0|2|220|203
165.28386|0|0|220|205
```

Listing 3.2: Monkey events

```
touch down 159|458
sleep 2.613
touch move 159|456
sleep 2.408
touch move 160|455
sleep 1.1767
touch up 160|455
sleep 15591.921
touch down 167|366
sleep 27.459
touch move 167|365
sleep 2.072
touch move 168|371
sleep 2.852
touch up 168|371
sleep 4119.185
key down 158
sleep 240.965
key up 158
sleep 4578.552
touch down 210|211
sleep 4.048
touch move 209|210
sleep 57.212
touch move 207|207
sleep 33.709
touch up 207|212
sleep 42763.863
key down 158
sleep 143.700
key up 158
sleep 9159.919
touch down 221|202
sleep 40.063
touch move 220|202
sleep 39.048
touch move 220|203
sleep 28.450
touch up 220|205
```

We explain this point using a real case example. We take, from our repository, two malware samples from BaseBridge family (**Listing 3.3** and **3.4**) and run them on the Android emulator: **Figure 3.11** shows that their layout is very similar, but the central button of the second sample is slightly shifted down.

Listing 3.3: BaseBridge first sample info

```
package_name: com.keji.danti207
```

Figure 3.11: Layout comparison of two similar applications.

```
version_name: 15
version_code: 2.4
sha1: 73bb65b2431fefd01e0ebe66582a40e74928e053
```

Listing 3.4: BaseBridge second sample info

```
package_name: com.zhangling.danti275
version_name: 19
version_code: 3.0.1
sha1: 00c154b42fd483196d303618582420b89cedbf46
```

If we try to merely re-run on the second sample the sequence of input events recorded during the original testing of the first sample, the situation shown in **Figure 3.12** may occur. We can see that the user taps on the button during the test, the VNC server records the coordinates of the tap and then monkey injects the touch event in the same point during test re-execution: the problem is that in that point there is not any button and the event is lost, invalidating the rest of execution.

To resolve this problem, we need more information about the view that has been stimulated during the original test: more precisely, we want a method that allows us to know which view has consumed the input event during the original test and a method to find that same view during test re-execution.

In *Section A.4* we introduced Android ViewServer: using this tool we can retrieve all the information regarding the views displayed on the screen. Combining this information with the coordinates of the touch events generated by the user, we can identify the sequence of view objects that have been

Figure 3.12: Failure example of a monkey-based test re-run.

stimulated during the test. Unfortunately, as we mentioned in *Section A.4*, the `DUMP` command used to retrieve this information is quite slow and it cannot be realistically used during an interactive test session. Luckily, we are not the only ones interested in a speed-up of ViewServer performance: as a matter of fact, we found an open-source project created to make ViewServer really usable for testing purposes[6]. The authors implemented a new dump command, named `DUMPQ`, that is 20x to 40x faster than the original `DUMP` command. To reach this result, it has been necessary to avoid the use of introspection as much as possible: therefore the authors modified the original Android system code in order to directly integrate the dump information inside the view objects, avoiding in this way the need of using introspection. Moreover, since we are interested only in the information needed to locate a view object on the screen, we further modified their code to only supply the information useful to our purposes.

Loading the patched Android system image at the emulator startup, we can now retrieve the view hierarchy displayed on the screen at run-rime in a reasonable time. We then further extended our VNC server implementation in order to perform the following steps when a new input event is received:

1. Process *PointerEvent* message.
2. Retrieve currently displayed window sending `GET_FOCUS` command to ViewServer.
3. Retrieve view hierarchy of the window sending `DUMPQ` command to ViewServer.

[6]Android testing patches project page: `https://code.google.com/p/android-app-testing-patches/`

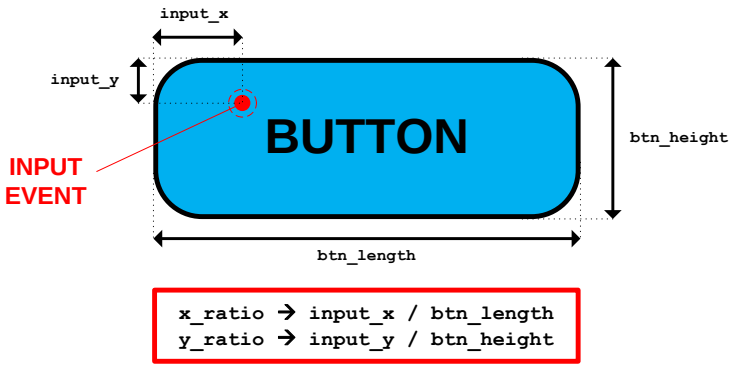

Figure 3.13: Input event relative position in respect to view object.

4. Search in the view hierarchy the deepest and rightmost view object (the last view drawn by SurfaceFlinger in case of overlapping views) containing the coordinates of the input event.

5. Save on file the path to the previously found view node.

At the end of test execution, we have now two files that store information about UI stimulation: a file containing the sequence of input events generated by the user and a file containing the sequence of path to the views that consumed those input events. As additional information, when we retrieve the view that consumed an event, we also keep trace of the relative position of the input event in respect to view (as shown in **Figure 3.13**): this information will be useful during test re-run.

The files generated by the VNC server are given as input to the PuppetDroid ReRunner tool, whose execution follows the pseudo-code shown in **Listing 3.5**:

Listing 3.5: PuppetDroid ReRunner pseudo-code.

```
1  INPUT: event_list_file, path_to_view_list_file;
2
3  // Extract lists from files
4  event_list = process_file(event_list_file)
5  path_to_view_list = process_file(path_to_view_list_file)
6
7  // The number of elements in each list must be the same
8  if len(event_list) != len(path_to_view_list)
9    abort()
10 endif
11
12 // Process the events
13 i = 0
14 while i < length(event_list)
15   ev = event_list[i]
16   if ev.type == 'TOUCH'
```

```
17      if ev.action == 'DOWN'
18        view_path = path_to_view_list[i]
19        original_x_pos = ev.x_pos
20        original_y_pos = ev.y_pos
21        x_ratio = view_path.x_ratio
22        y_ratio = view_path.y_ratio
23
24        // Get the view hierarchy of the currently
25        // displayed window using viewserver
26        view_hierarchy = get_focused_window_hierarchy()
27
28        // Search in the hierarchy the view that
29        // consumed input event in the original test
30        view = find_view_in_hierarchy(view_hierarchy, view_path)
31        if view == NULL
32          abort()
33        endif
34
35        // Calculate new event coordinates
36        new_x_pos = view.x_pos + (view.width * x_ratio)
37        new_y_pos = view.y_pos + (view.height * y_ratio)
38
39        // calculate the deviation from the
40        // original coordinates
41        x_deviation = new_x_pos / original_x_pos
42        y_deviation = new_y_pos / original_y_pos
43
44        // Update ev coordinates and inject it
45        ev.x_pos = new_x_pos
46        ev.y_pos = new_y_pos
47        inject_touch_event(ev)
48
49        // Apply deviation to the following events
50        // until a touch up event is found
51        i = i + 1
52        ev = event_list[i]
53        while not(ev.type == 'TOUCH' and ev.action == 'UP')
54          if ev.type == 'TOUCH'
55            original_x_pos = ev.x_pos
56            original_y_pos = ev.y_pos
57            ev.x_pos = original_x_pos * x_deviation
58            ev.y_pos = original_y_pos * y_deviation
59            inject_touch_event(ev)
60          elif ev.type = 'KEY'
61            inject_key_event(ev)
62          elif ev.type = 'WAIT'
63            execute_wait_event(ev)
64          else
65            abort()
66          endif
67          i = i + 1
68          ev = event_list[i]
69        endwhile
70
```

```
71        // Process touch up event
72        if ev.type == 'TOUCH' and ev.action == 'UP'
73          original_x_pos = ev.x_pos
74          original_y_pos = ev.y_pos
75          ev.x_pos = original_x_pos * x_deviation
76          ev.y_pos = original_y_pos * y_deviation
77          inject_touch_event(ev)
78        endif
79    elif ev.type = 'KEY'
80      inject_key_event(ev)
81    elif ev.type = 'WAIT'
82      execute_wait_event(ev)
83    else
84      abort()
85    endif
86    i = i + 1
87  endwhile
```

As shown, the ReRunner automatically adapts the original coordinates of input events in order to fit them to the layout of the similar APK sample, avoiding in this way event loss due to small layout changes. Unfortunately, during our tests, we found out that this approach can still fail in some cases: it could happen that the view that receives the input event is not the view that eventually consumes it.

To explain this passage, we have first to briefly introduce how Android handles touch events. When a touch event is generated by the system, the `Activity.dispatchTouchEvent()` method of the currently running Activity is called. This method dispatches the event to the root view in the hierarchy and waits for the result: if no view consumes the event, the Activity calls `onTouchEvent()` in order to consume itself the event before terminating. When a View object receives a touch event, the `View.dispatchTouchEvent()` is called: this method first tries to find an attached listener to consume the event, calling `View.OnTouchListener.onTouch()`, then tries to consume the event itself calling `View.onTouchEvent()`. If there is neither a listener nor the `onTouchEvent()` method is implemented, the event is not consumed and it flows back to the parent. When a ViewGroup receives a touch event, it iterates on its children views in reverse order and, if the touch event is inside the view, it dispatches the event to the child. If the event is not consumed by the child, it continues to iterate on its children until a view consumes the event. If the event is not consumed by any of its children, the ViewGroup acts as a View and tries to consume itself the event. Eventually, if it is not able to consume the event it sends back to the parent. A more detailed description on Android Touch System works can be found here [39].

Figure 3.14 shows a couple of examples of touch events management: in the first case, the event flows down through the hierarchy, and since it is not

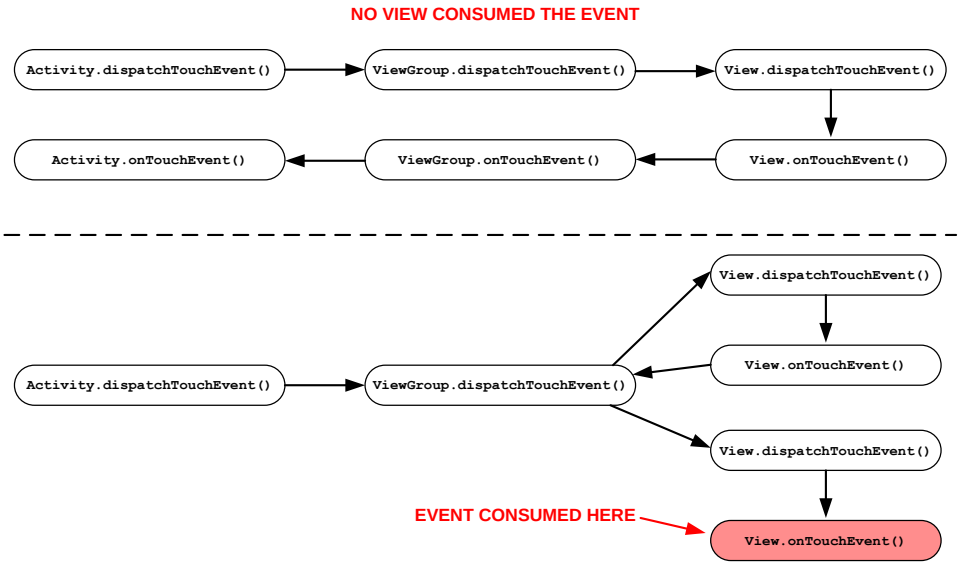

Figure 3.14: Examples of touch event management in Android.

consumed by any view, it comes back to the Activity. In the second case, the event is consumed by the second View child of the ViewGroup object: this is the case that sometimes can cause problems to our system. To face this issue, we slightly modified Android source code in order to log which activity has consumed the touch event. Moreover, we modified the VNC server in order to store the path to all the deepest nodes in the hierarchy that can consume the touch event. Finally we combine the information thus collected to extract the list of the paths to the views that actually consumed the touch events. In this way we succeeded in obtaining a very robust method to record and re-execute UI stimulation on similar application.

3.3.7.2 Similarity

In this section we present some implementation details on how we calculate the similarity between two Android samples.

In *Section 6.2* we describe some theoretical approaches addressing this topic. Between them, we choose to use the one that leverages *Normalized Compressed Distance* (NCD) in order to approximate Kolmogorov complexity. This approach has been implemented in androsim, a tool included in Androguard[7], a widely used suite of python scripts used to analyze, decompile, reverse and manipulate Android applications. androsim is the tool in the suite used to compare two Android APKs, in order to find out similarities

[7]Androguard project page: `https://code.google.com/p/androguard/`

and differences between them. Given two application as input, it provides as result:

- the identical methods;
- the similar methods;
- the deleted methods;
- the new methods;
- the skipped methods;
- a similarity score (between 0.0 and 100.0).

To calculate this values, androsim retrieves the list of methods of the two samples decompiling them, then it removes the identical methods basing on a hashing comparison, filters a set of elements labeled as "skippable", such as methods from known libraries, and finally uses NCD on remaining methods to find similarity associations. Basing on the results obtained it calculates a final similarity score. **Figure 3.15** shows androsim basic workflow.

For the computation of sample similarities, we dedicated a separated machine, accessible by PuppetDroid Main Server through a set of restful APIs. During our tests we encountered some problems in the use of androsim.

The first one is that it seems not to be symmetrical: inverting the order in which the two samples are passed as parameters leads to different results. This is probably due to an internal implementation error: unfortunately we did not succeed in finding it. To face this problem, we decided that two samples are similar over a threshold T only if androsim provides a similarity score greater than T for both the way the two samples can be compared. At the moment we consider similar two applications whose similarity score is greater than 80%.

The second problem is related to the computation time needed for a comparison: it is usually in the order of tens of seconds, even if sometimes it may require some minutes. Using a quad-core machine as similarity server and exploiting parallelization, we can reach an average time of about 11s. The problem is that, when a new sample is uploaded on our repository, we should be able to calculate the similarity score between that sample and the whole sample repository in a reasonable time. Moreover, following this approach, the growth of the repository also leads to a linear growth of the computation time needed to calculate the similarity score of a new sample. To make an example, our sample repository contains more than 7000 APKs: in order to compare a single APK with the other samples in the repository, the time required is:

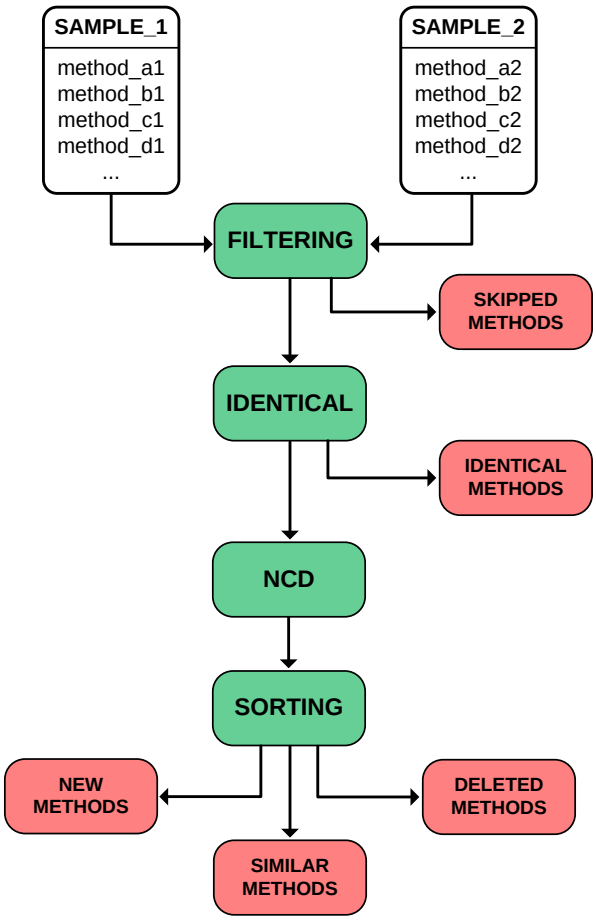

Figure 3.15: androsim basic workflow.

$$computation\_time = 2 * \#\_samples * avg\_comp\_time = 2 * 7000 * 11s = 154000s \approx 42h$$

As we can see, it is a huge time to compare a single sample with the whole repository. In *Section 6.2* we present alternative methods that propose quick and scalable strategies to calculate similarity on great set of applications. Unfortunately these techniques have not been implemented in publicly available tools yet, so, due to time constraints, current implementation of PuppetDroid relies on androsim. However, this is only a temporary solution and we are planning to integrate in our system a scalable and fast method to calculate application similarity.

Chapter 4

Experimental evaluation

In this chapter we present the experimental tests performed in order to evaluate our approach. In particular, our experimental evaluations can be divided in two sets: with the first one, presented in *Section 4.1*, we wanted to verify the effectiveness of PuppetDroid UI stimulation, while the second one, presented in *Section 4.2*, has been conducted with the aim of testing the feasibility of our re-run approach.

4.1 PuppetDroid stimulation evaluation

We recall that PuppetDroid aids dynamic analysis of Android (malicious) applications. PuppetDroid collects dynamic behaviors data about the APK samples, which should reach better coverage than automatic methods. First, we verify that our stimulation approach leads to a better stimulation compared to other automatic analysis approaches. For this, we compare the number of behaviors exercised with PuppetDroid with the number of behaviors exercised with automatic approaches, namely Monkey, used in some dynamic malware analysis framework such as Andrubis [25], and the system events stimulation strategy proposed in CopperDroid [37].

4.1.1 Dataset

We conducted our experimental evaluation on a set of 10 APK samples: 8 of them are malware samples provided by the Android Malware Genome Project [48] while the other 2 are goodware samples retrieved from Google Play. Specific details on each sample used in our tests are reported in **Table 4.1**.

Malware Samples	
Pagkage Name	com.keji.danti207
Version	15
SHA1	73bb65b2431fefd01e0ebe66582a40e74928e053
Malware Family	BaseBridge
Pagkage Name	com.keji.danti80
Version	13
SHA1	58f2bcf5811fcde82172d7e1e2faba25c5c75edd
Malware Family	BaseBridge
Pagkage Name	com.zhangling.anTest20
Version	10
SHA1	7c0af89dd083b97db3dd70b7a2329c4a21a2c592
Malware Family	DroidKungFu
Pagkage Name	com.tutusw.onekeyvpn
Version	7
SHA1	98b83e122178ebe9e6b43aaec09af4661a5e92ec
Malware Family	DroidKungFu
Pagkage Name	com.atools.cuttherope
Version	5
SHA1	64013d749086e90bdcfccb86146ad6e62b214cfa
Malware Family	DroidKungFu
Pagkage Name	HamsterSuper.Client.Game
Version	2
SHA1	aa9216c96ab29477966d0ff83b0326241f733734
Malware Family	YZHC
Pagkage Name	HamsterSuper.Client.Game
Version	1
SHA1	593dd0ec6d6b9802d5d4626978cead4c83051b4a
Malware Family	YZHC
Pagkage Name	com.systemsecurity6.gms
Version	1
SHA1	c9368c3edbcfa0bf443e060f093c300796b14673
Malware Family	Zitmo
Goodware Samples	
Pagkage Name	jp.sblo.pandora.jota
Version	81
SHA1	437a1f8059c3458fa2d1f4a1d902bbeefae9e8a9
Pagkage Name	com.WhatWapp.Briscola
Version	30
SHA1	d32b99235a6c9effa6e69cff523732c1fbc964b8

Table 4.1: Dataset used to compare stimulation approaches.

The dataset we used is quite small because we performed multiple tests on each sample and the output of each test has been manually inspected in order to examine the differences between different approaches. Therefore, we preferred focus on a small dataset in order to perform a deeper analysis of each test result.

4.1.2 Experimental setup

First of all, in order to carry on this experimental evaluation, we need an effective way to compare two stimulation approaches. We then decided to leverage the comparison method proposed by CopperDroid's authors in their work [37]:

- Use the CopperDroid instrumented emulator to collect system call traces during test execution.
- Semantic analysis of system call traces collected in order to extract a list of the stimulated behaviors.
- Use CopperDroid libraries to compare the behavior lists extracted from two test executions.

In this way, given two test executions, we can establish the total stimulated behaviors in each test, the behaviors stimulated by both the tests (intersection) and the behaviors stimulated only by either test (set difference).

A single test instance consists of the following sequence of actions:

1. Create and start a clean CopperDroid sandbox.
2. Install the APK sample on the sandbox.
3. Perform the selected stimulation approach.
4. Retrieve CopperDroid result files and delete the AVD instance.

For each sample in the experimental dataset, we performed the following tests:

- 1 test without stimulation (labeled as *NoStim*).
- 1 test using CopperDroid stimulation strategy (labeled as *Copper*).
- 20 tests using Monkey stimulation (labeled as *Monkey*):
  - 5 tests injecting 500 input events;
  - 5 tests injecting 1000 input events;
  - 5 tests injecting 2000 input events;

  - 5 tests injecting 5000 input events.

- 1 test using PuppetDroid approach (labeled as *Puppet*).

In particular, for both the test NoStim and Copper, we started the application, waited 30 seconds and finally terminated the execution.

Since Monkey sometimes hangs, for each monkey test we set a timeout of 10 minutes, thereafter the test is terminated.

PuppetDroid tests have been executed using a HTC Wildfire S as user's device.

Finally, as previously said, we leveraged CopperDroid libraries to implement a couple of scripts in order to compare behavior lists of two test executions.

4.1.3 Results

We present here the results of our experimental evaluations. First, we introduce the terminology used to label the results:

Total bhvs – average number of behaviors per test observed by CopperDroid. It can include duplicate behaviors.

Distinct bhvs – average number of distinct behaviors per test observed by CopperDroid (i.e., without duplicates).

Exclusive bhvs of A in respect to B – average number of behaviors observed by CopperDroid only in test exercised by A and not observed in test exercised by B.

For each exercising approach, we extracted the total behaviors stimulated: **Figure 4.1** provides a graphical overview of this first analysis. The blue bars indicate total behaviors per test while the red bars indicate distinct behaviors.

Stimulated Behaviors	
No Stimulation	14.7 (7.1 distinct)
Monkey	17.925 (6.69 distinct)
CopperDroid	19.0 (10.6 distinct)
PuppetDroid	46.8 (16.4 distinct)

Table 4.2: Summary of the results obtained in the experimental evaluation of PuppetDroid stimulation approach.

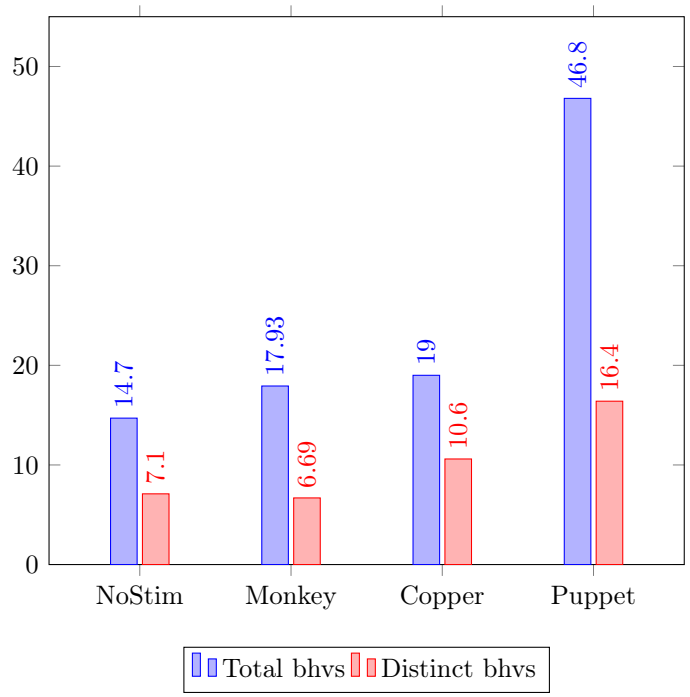

Figure 4.1: Total behaviors per test.

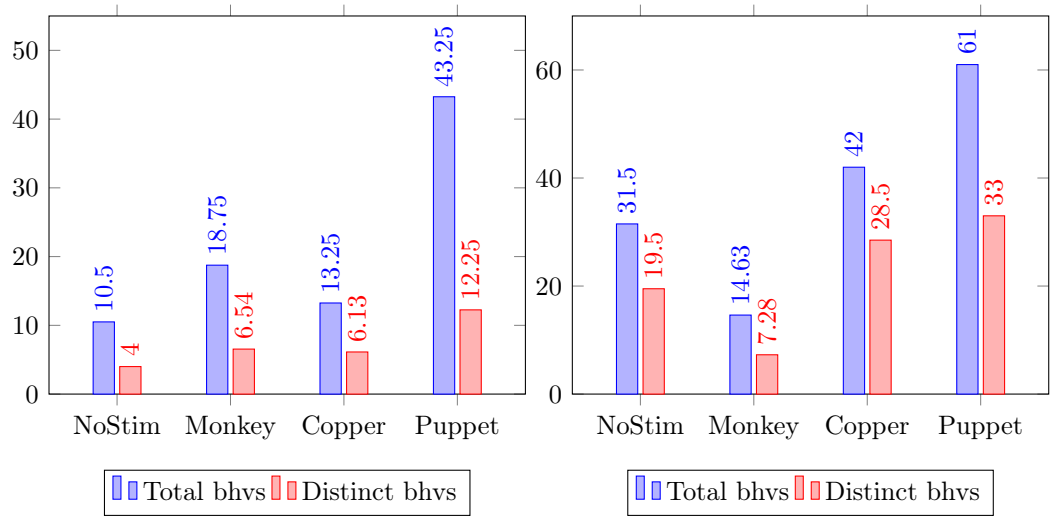

Figure 4.2: Total behaviors per test for malware (on the left) and goodware (on the right) samples.

As we expected, PuppetDroid human-driven stimulation succeeds in stimulating more behaviors than automatic approaches: we are able to stimulate 254% of total behaviors and 200% of distinct behaviors more than the automatic stimulation methodologies. To make result presentation more clear, we report behavior data also in table **Table 4.2**. In **Figure 4.2** we provide the same information, but dividing the results between the tests performed on malware and on goodware samples.

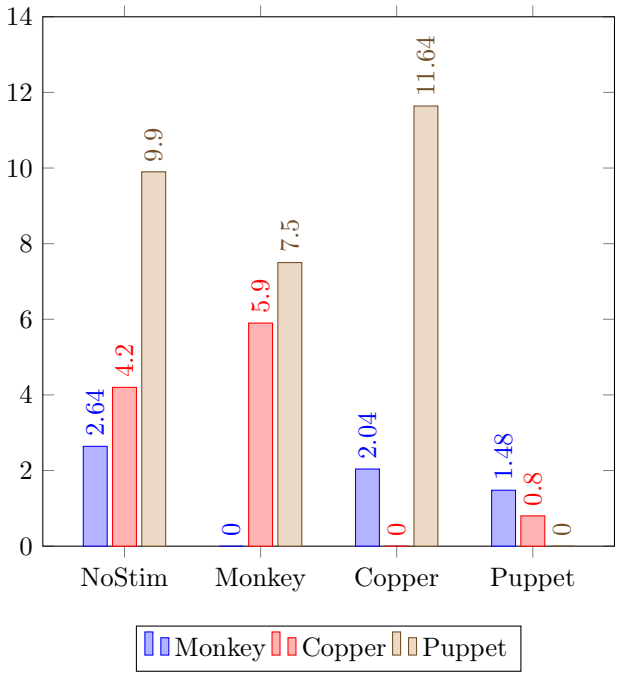

Figure 4.3: Comparison of exclusive behaviors stimulated with a stimulation approach in respect to the others.

The data collected from CopperDroid result files allow us to extract another interesting information: the exclusive behaviors stimulated by only one approach compared to another, that is the number of behaviors that are only stimulated with the first approach but not with the second one. **Figure 4.3** shows this information: as we can see, the exclusive behaviors of Monkey and CopperDroid (identified by the bars in blue and red) are fewer than the ones of PuppetDroid. In particular, considering the exclusive behaviors generated by the three approaches compared with the NoStim test, we have that PuppetDroid is able to stimulate 375% exclusive behaviors more in respect to Monkey and 235% more i respect to CopperDroid. We also illustrate comparison data in **Table 4.3**, in order to make result presentation more clear. As before, we report also the data regarding only malware and goodware (**Figure4.4**) samples.

Only in \ in respect to	NoStim	Monkey	Copper	Puppet
Monkey	2.64	0	2.04	1.48
Copper	4.2	5.9	0	0.8
Puppet	9.9	7.5	11.64	0

Table 4.3: Comparison of exclusive behaviors stimulated with a stimulation approach in respect to the others..

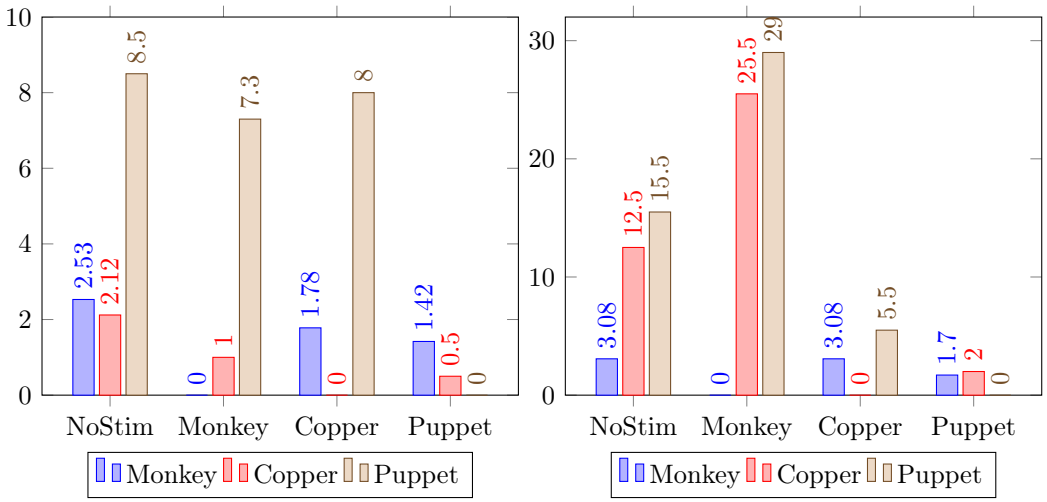

Figure 4.4: Comparison of exclusive behaviors stimulated with a stimulation approach in respect to the others, considering only malware (on the left) and goodware (on the right)samples.

This data confirm our first assumption: PuppetDroid UI stimulation approach allows to obtain better results in respect to automatic approaches used in other dynamic analysis frameworks.

Finally, we point up a particular case where PuppetDroid succeeded in stimulating a malicious behavior that the other stimulation strategies did not exercised. The malware sample under analysis is `com.keji.danti80`, belonging to BaseBridge malware family. BaseBridge is a trojan that, once installed, prompts the user with an upgrade dialog: if users accept to do so, the malware will install a malicious service on the phone. This service communicates with a control server to receive instructions to perform malicious actions (e.g., place calls or send messages to premium numbers). Meanwhile, the malware also blocks messages from the mobile carrier in order to prevent users from getting fee consumption updates: in this way all malicious activities are undertaken stealthily without the users' knowledge or consent. More details on this malware can be found in [34, 31, 41] .

Behavior	Blob	Hit
write	{'filename': u'/sys/qemu_trace/process_name'}	2
write	{'filename': u'/sys/qemu_trace/process_name'}	2
write	{'filename': u'/sys/qemu_trace/process_name'}	2
write	[MKDIR] /data/data/com.keji.danti80/files	1
write	{'filename': u'/data/data/com.keji.danti80/files/xxx.apk'}	1
write	{'filename': u'/sys/qemu_trace/process_name'}	2
write	{'filename': u'/sys/qemu_trace/process_name'}	2
write	[MKDIR] /data/data/com.sec.android.bridge/shared_prefs	1
write	[UNLINK] /data/data/com.keji.danti80/files/xxx.apk	1
connect	{'host': '10.0.2.3', 'retval': 0, 'port': 53}	1
outgoing_dns_query	{'query_data': 'b3.8866.org. 1 1'}	1
connect	{'host': '221.5.133.18', 'retval': -115, 'port': 8080}	1
write	{'filename': u'/data/data/com.sec.android.bridge/shared_prefs/first_app_perferences.xml'}	3
write	[UNLINK] /data/data/com.sec.android.bridge/shared_prefs/first_app_perferences.xml.bak	2
write	{'filename': u'/data/data/com.keji.danti80/files/atemp.jpg'}	1
write	[UNLINK] /data/data/com.keji.danti80/files/atemp.jpg	1
write	{'filename': '221.5.133.18 port:8080'}	2

Table 4.4: List of behaviors extracted testing `com.keji.danti80` malware sample.

Analyzing the sample with PuppetDroid we obtained the list of behaviors shown in **Table 4.4**. The red colored lines indicate a behavior that none of the other stimulation techniques were able to reveal. The malware writes another APK file, `xxx.apk`, on the filesystem. As a matter of fact, during the test, the application prompts the user to install a new application, named BridgeProvider, to complete the update, as shown in **Figure 4.5**.

BridgeProvider is the service used by BaseBridge trojan to accomplish its malicious behaviors, as it can seen by analyzing the APK with the *Android Asset Packaging Tool* (aapt). The extracted information is shown in **Listing 4.1**.

Listing 4.1: Static information extracted from xxx.apk payload

```
application_name: BridgeProvider
package_name: com.sec.android.bridge
version_name: 13
version_code: 4.2
sha1: 9dd5052b09f9b82eecaedc5d02391cbe0e8e515e
used_permissions:   android.permission.READ_SMS
                    android.permission.RECEIVE_SMS'
```

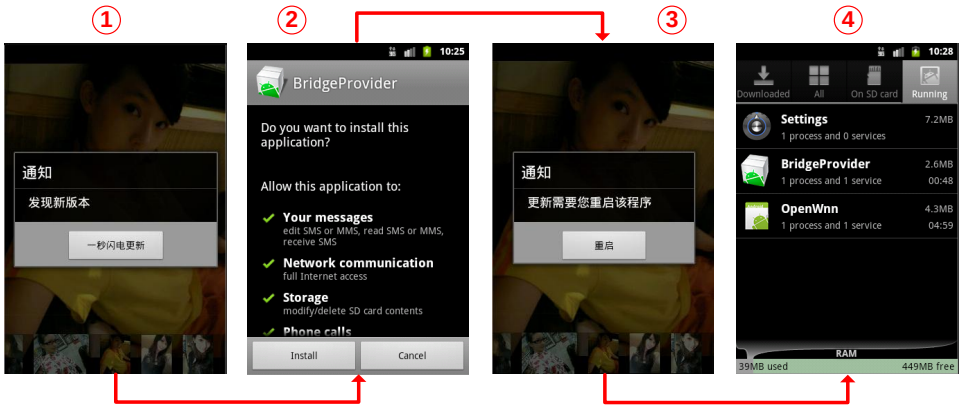

Figure 4.5: Steps to install BridgeProvider payloads: 1) Ask for application update; 2) Install payload; 3) Restart application; 4) Malicious service running on device.

```
                    android.permission.SEND_SMS'
                    android.permission.READ_PHONE_STATE'
                    android.permission.INTERNET'
                    android.permission.WRITE_SMS'
                    android.permission.RECEIVE_BOOT_COMPLETED'
                    android.permission.VIBRATE'
                    android.permission.WRITE_EXTERNAL_STORAGE'
                    android.permission.READ_EXTERNAL_STORAGE'
```

In conclusions, the other stimulation approaches did not exercised the malware such that it revealed its true malicious behavior, with the consequent risk to consider the sample as safe. Instead, using PuppetDroid, the analyst is able to detect such a potential dangerous behavior and subsequently analyze in detail the functioning of the application.

Conclusions The results of the experiments confirmed our intuition that automatic UI stimulation approaches can only exercise a subset of the (malicious) behaviors of a malware during dynamic analysis. On the other hand, PuppetDroid approach based on human-driven UI exercising allows to reproduce typical victim interaction with the malware and to reach then higher code coverage.

4.2 PuppetDroid scalability evaluation

To make our approach scale, PuppetDroid leverages the re-execution of previously recorded UI stimulation traces on applications with a similar layout. As explained in *Section 2.3*, our idea is based on 2 assumptions:

- Many malware samples are actually repackaged versions of other samples.
- If we succeed in exercising the (malicious) behaviors in a sample, we exercise the same behaviors, or behaviors triggered by the same UI stimulation, in similar samples.

This section presents the tests performed in order to verify these assumptions.

4.2.1 Dataset

For this experimental evaluation we took 4 APK samples from the Android Malware Genome Project [48]. Then, we exploited Androguard to retrieve similar APKs from a repository of over 7000 samples taken from the Android Malware Genome Project, Google Play and alternative Android markets. As mentioned in *Section 3.3.7.2*, Androguard similarity tool is not very fast in calculating the similarity score, so we calculated our similarity comparisons only on a subset of our sample repository. This is a temporary limitation of current implementation of PuppetDroid: in fact, as explained in *Section 7*, we are going to substitute androsim with a scalable and fast application similarity strategy. Specific details on the 4 samples originally stimulated with PuppetDroid are reported in **Table 4.5**.

Malware Samples	
Pagkage Name	com.keji.danti207
Version	15
SHA1	73bb65b2431fefd01e0ebe66582a40e74928e053
Malware Family	BaseBridge
Pagkage Name	com.tutusw.onekeyvpn
Version	7
SHA1	98b83e122178ebe9e6b43aaec09af4661a5e92ec
Malware Family	DroidKungFu
Pagkage Name	HamsterSuper.Client.Game
Version	2
SHA1	aa9216c96ab29477966d0ff83b0326241f733734
Malware Family	YZHC
Pagkage Name	com.keji.danti160
Version	14
SHA1	87f1eb6baa69745d306e8765bb085621410c241f
Malware Family	BaseBridge

Table 4.5: Dataset used to validate our re-run approach.

4.2.2 Experimental setup

To verify if our re-execution approach is feasible, we need a way to evaluate the results of the re-executed tests. We follow three methods:

ManualVsRe-exec – Compare the behaviors exercised with the manual stimulation technique with the behaviors extracted from the re-executed tests.

Re-execBhvs – Verify if an interesting malicious behavior stimulated in the original sample is also stimulated during the re-execution on a similar application.

AutomaticVsRe-exec – Compare the behaviors exercised in the re-executed tests with the behaviors extracted using automatic stimulation tools, as done in the first set of experimental evaluations.

These two evaluation approaches provide different information: with the first one we can know if we succeed in stimulating interesting malicious behaviors, seen in the original sample, also in the re-executed tests, whereas the second one indicates if the original stimulation applied on a different application allows us to still obtain a better stimulation compared with the automatic stimulation approaches.

We structured each test as follows, for each of the 4 APKs:

- Use PuppetDroid to manually test the application: the system automatically stores information related to UI stimulation during test execution.
- Use `androsim` to search APKs similar to the one previously tested: we consider two samples as similar if their similarity score is greater than 80%.
- Leverage PuppetDroid framework to automatically re-execute previously recorded UI stimulation on similar applications.
- Test similar applications with automatic stimulation approaches:
  - 1 test without external stimulation
  - 20 tests using Monkey stimulation
  - 1 test using CopperDroid stimulation strategy.
- Perform the two evaluation strategies explained before.

For each sample in the dataset presented in **Table 4.5**, we performed one test, except for `com.keji.danti207` sample that we used in two tests (the reason for this choice will be explained in *Section 4.2.3*).

PuppetDroid and automatic stimulation tests have been executed as previously described in *Section 4.1.2*, using a HTC Wildfire S as the user device.

Finally, for each test re-execution, we also calculated a re-execution score, expressed as a percentage, that indicates how many UI events have been successfully re-executed. For example, suppose we have a recording of a UI stimulation with 20 events: if our `puppet_rerunner` tool succeeds in re-executing 10 UI events but it is not able to find the correct view to inject the 11^{th} event, the re-execution is terminated. We then have a re-execution score of 50%.

4.2.3 Results

ManualVsRe-exec We first show the results of the comparison between the behaviors retrieved using the manual stimulation of the original sample and the behaviors extracted from similar applications stimulated with the re-execution of the same stimulation.

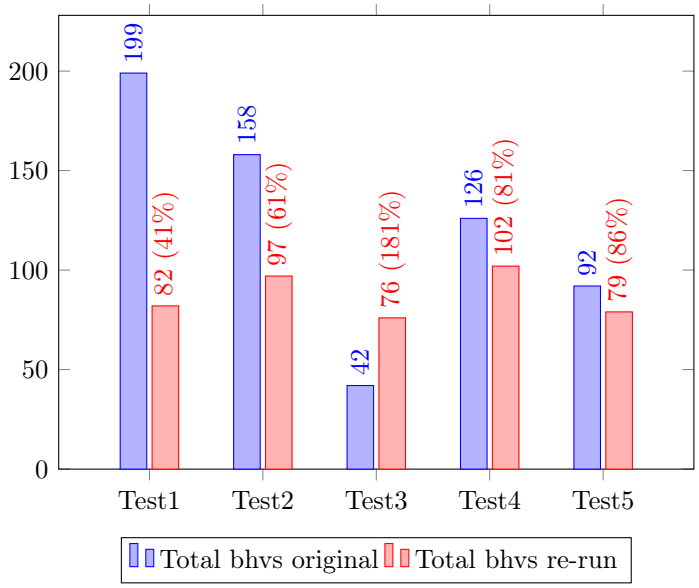

Figure 4.6: Comparison of total behaviors stimulated in the original execution vs. the average total behaviors stimulated in re-executed tests.

In particular, **Figure 4.6** shows the comparison between the total number of behaviors stimulated in the original test (in blue) in respect to the average of the ones stimulated in re-executed tests (in red), **Figure 4.7** shows the comparison of distinct behaviors, while **Figure 4.8** shows the exclusive be-

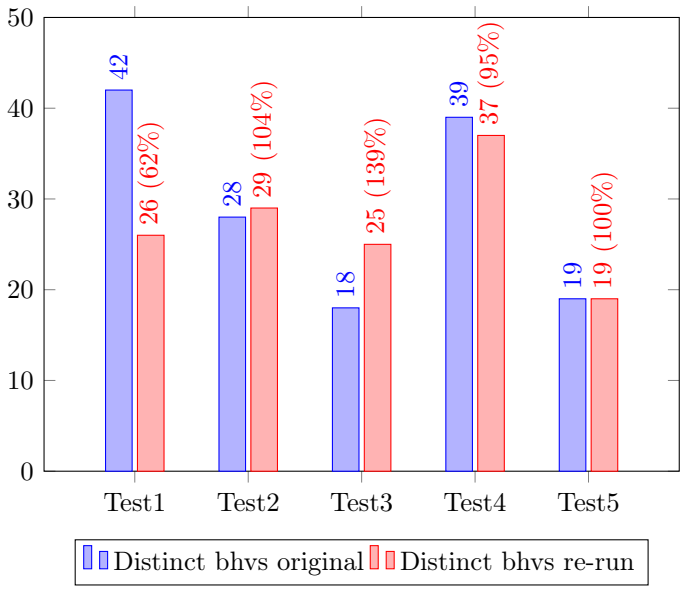

Figure 4.7: Comparison of distinct behaviors stimulated in the original execution vs. the average distinct behaviors stimulated in re-executed tests.

haviors stimulated only in the original test (always in blue) in respect to the exclusive behaviors stimulated on average only in re-executed tests.

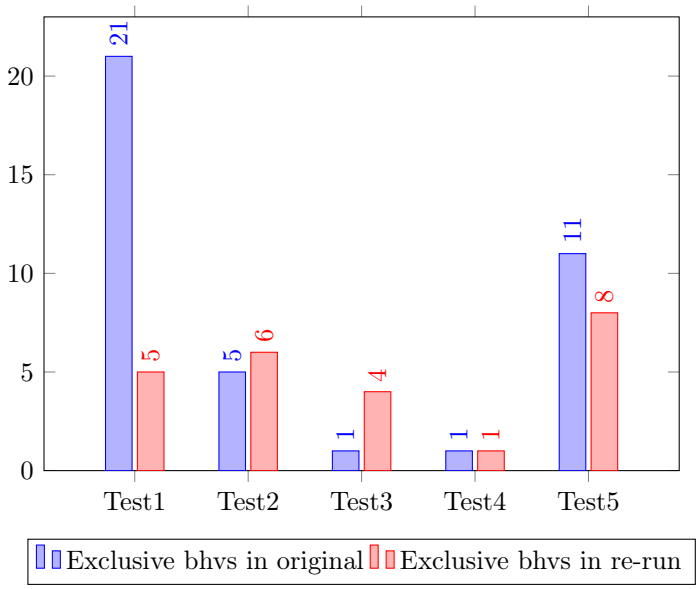

Figure 4.8: Comparison of exclusive behaviors stimulated in the original execution vs. the average exclusive behaviors stimulated in re-executed tests.

At first one would expect that the behaviors extracted in the original test are

more than the ones stimulated in re-executed tests. In some cases it is not true, (e.g., in *Test3*). This is due to the fact that we are comparing the behaviors exercised in different, even if similar, applications: it is possible that an application similar to the one originally tested, generates more behaviors even if less stimulated. To make an example, it is possible that when an application A is started it generates 10 behaviors, while when an application B, similar to A, is started it generates 20 behaviors: the same holds also for the UI stimulation, so clicking on a button of A we have 2 behaviors, while clicking on the same button on B leads to 4 behaviors.

Re-execBhvs We now verify if an interesting malicious behavior stimulated in the original sample is also stimulated during the re-execution on a similar application. To do so we consider the application `com.keji.danti160`, belonging to BaseBridge malware family: information about this sample are shown in **Table 4.5**, while experimental data related to this test are labeled as *Test5* in the diagrams. We chose this sample because during the test it showed a behavior similar to the one shown by `com.keji.danti80` sample described in *Section 4.1.3*: when started, the application asks the user to update it and installs a malicious service, named `BridgeProvider`, on the phone. This behavior is shown in **Figure 4.9**, while the list of behaviors extracted with CopperDroid during the test is presented in **Table 4.6**: red lines indicates the malicious actions executed by the application.

Figure 4.9: Another example of BaseBridge malware: 1) Ask for application update; 2) Install payload; 3) Restart application; 4) Malicious service running on device.

Scanning our sample repository with androsim, we found one sample, named `com.keji.danti161`, that produces a similarity score greater than 80%: information related to this sample is presented in **Table 4.7**, while a comparison of the layout of the two samples is shown in **Figure 4.10**. As we can see, the two applications seem to be identical, even if they have a different pack-

Behavior	Blob	Hit
write	{'filename': u'/sys/qemu_trace/process_name'}	2
write	{'filename': u'/sys/qemu_trace/process_name'}	2
write	{'filename': u'/sys/qemu_trace/process_name'}	2
write	[MKDIR] /data/data/com.keji.danti160/shared_p refs	1
write	[MKDIR] /data/data/com.keji.danti160/files	1
write	{'filename': u'/data/data/com.keji.danti160/files/xxx.apk'}	1
write	{'filename': u'/sys/qemu_trace/process_name'}	2
write	{'filename': u'/sys/qemu_trace/process_name'}	2
write	[MKDIR] /data/data/com.sec.android.bridge/shared_prefs	1
connect	{'host': '10.0.2.3', 'retval': 0, 'port': 53}	1
outgoing_dns_query	{'query_data': 'b3.8866.org. 1 1'}	1
write	[UNLINK] /data/data/com.keji.danti160/files/xxx.apk	1
write	[MKDIR] /data/data/com.keji.danti160/databases	1
write	{'filename': u'/data/data/com.keji.danti160/databases/db.db'}	24
write	{'filename': u'/data/data/com.sec.android.bridge/shared_prefs/first_app_perferences.xml'}	3
write	[UNLINK] /data/data/com.sec.android.bridge/shared_prefs/first_app_perferences.xml.bak	2
connect	{'host': '221.5.133.18', 'retval': -115, 'por t': 8080}	2
write	{'filename': u'/data/data/com.keji.danti160/shared_prefs/com.keji.danti160.xml'}	22
write	[UNLINK] /data/data/com.keji.danti160/shared_prefs/com.keji.danti160.xml.bak	21

Table 4.6: List of behaviors extracted testing `com.keji.danti160` malware sample.

age name and a different hash value: this is a typical example of malware repackaging.

Pagkage Name	com.keji.danti161
Version	14
SHA1	b457113c46b19dcec6ebb68efe24f1460237389d
Malware Family	BaseBridge

Table 4.7: Information related to `com.keji.danti161` sample.

Re-executing the UI stimulation recorded with PuppetDroid on the application `com.keji.danti161`, we extracted the list of behaviors shown in **Table 4.8**: red colored lines present the same malicious actions stimulated in the original test execution. This example illustrates that re-execution approach is valid: in fact, we succeeds in automatically stimulating an interesting malicious behavior in an application leveraging application similarity, to discover repackaged samples, and UI stimulation re-execution.

Behavior	Blob	Hit
write	{'filename': u'/sys/qemu_trace/process_name'}	2
write	{'filename': u'/sys/qemu_trace/process_name'}	2
write	{'filename': u'/sys/qemu_trace/process_name'}	2
write	[MKDIR] /data/data/com.keji.danti161/shared_prefs	1
write	[MKDIR] /data/data/com.keji.danti161/files	1
write	{'filename': u'/data/data/com.keji.danti161/files/xxx.apk'}	1
write	{'filename': u'/sys/qemu_trace/process_name'}	2
write	{'filename': u'/sys/qemu_trace/process_name'}	2
write	[MKDIR] /data/data/com.sec.android.bridge/shared_prefs	1
write	{'filename': u'/data/data/com.sec.android.bridge/shared_prefs/first_app_perferences.xml'}	2
write	[UNLINK] /data/data/com.sec.android.bridge/shared_prefs/first_app_perferences.xml.bak	1
connect	{'host': '10.0.2.3', 'retval': 0, 'port': 53}	1
outgoing_dns_query	{'query_data': 'b3.8866.org. 1 1'}	1
write	[UNLINK] /data/data/com.keji.danti161/files/xxx.apk	1
connect	{'host': '221.5.133.18', 'retval': -115, 'por t': 8080}	1
write	[MKDIR] /data/data/com.keji.danti161/databases	1
write	{'filename': u'/data/data/com.keji.danti161/shared_prefs/com.keji.danti161.xml'}	17
write	[UNLINK] /data/data/com.keji.danti161/shared_prefs/com.keji.danti161.xml.bak	16
write	{'filename': u'/data/data/com.keji.danti161/databases/db.db'}	24
write	{'filename': u'/data/system/dropbox/drop68.tmp'}	4

Table 4.8: List of behaviors extracted from `com.keji.danti161` malware sample with UI re-execution.

Figure 4.10: Layout comparison of `com.keji.160` with `com.keji.161`.

AutomaticVsRe-exec We now evaluate the stimulation obtained with re-execution compared with the automatic stimulation approaches. Comparing the behaviors extracted from re-executed tests with the ones retrieved stimulating the same samples with Monkey and CopperDroid, we obtained the data shown in **Figure 4.11**. As we can see, the re-executed stimulation still allows to stimulate more behaviors in respect to automatic approaches: in fact, using PuppetDroid re-execution, we are able to stimulate 187% of total behaviors and 137% of distinct behaviors more than the automatic stimulation methodologies. Moreover, with re-execution we stimulate 276% exclusive behaviors more respect than Monkey and 312% more respect than CopperDroid. It is also worth noting that this is a conservative estimate of re-execution effectiveness: as a matter of fact, our experimental data contain also cases in which re-execution promptly failed after test beginning. We will examine these particular failure cases in the next lines. Finally, **Table 4.9** sums up the results obtained with these tests.

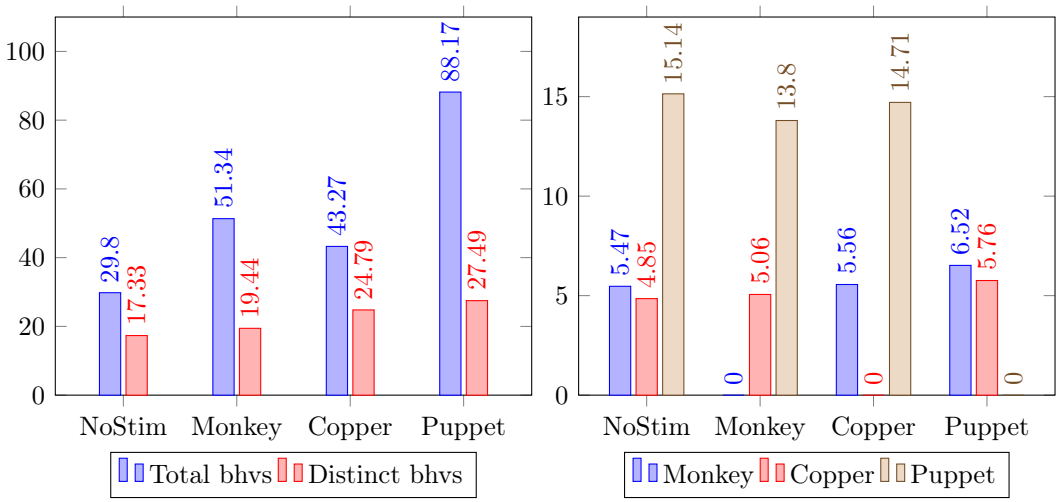

Figure 4.11: Comparison of behaviors stimulated with re-execution in respect to behaviors extracted using automatic stimulation: total and distinct behaviors on the left, exclusive behaviors on the right.

Re-execution failures

We now illustrate some particular cases that can make a test re-execution fail.

We observed the first case during the execution of the test labeled as *Test3*. In this test we found 6 samples similar to the originally tested application: re-executing the UI stimulation recorded with PuppetDroid on these samples, we noticed that 3 executions obtained a re-execution score of 100%, while

Overview	
Number of similar APKs	4.00
Re-execution score	60.62%
Comparison with original test	
Original test behaviors	123.00 (29.00 different)
Re-executed tests behaviors	88.17 (27.49 different)
Exclusive behaviors in original	7.00
Exclusive behaviors in re-executed	4.00
Comparison with automatic stimulation	
Stimulated Behaviors	
No Stimulation	29.80 (17.33 different)
Monkey	51.34 (19.44 different)
CopperDroid	43.27 (24.79 different)
PuppetDroid	88.17 (27.49 different)

Exclusive Behaviors				
Only in \ in respect to	**NoStim**	**Monkey**	**Copper**	**Puppet**
Monkey	5.47	0	5.56	6.52
Copper	4.85	5.06	0	5.76
Puppet	15.14	13.80	14.71	0

Table 4.9: Summary of the results obtained in the experimental evaluation of PuppetDroid re-execution approach (average values per test).

3 obtained a re-execution score of 0% (i.e., the re-execution promptly failed after test beginning).

Analyzing the layout of the samples for which the re-execution failed with the layout of the originally tested application, we found out that they considerably differ, as shown in **Figure 4.12**. Clearly, our `puppet_rerunner` tool cannot find the correct view to be stimulated in such a different layout and the test unavoidably fails.

So, the first cause for re-execution failure is due to a wrong identification of similar applications: our similarity comparison approach can consider two applications as similar, even if they greatly differ in the layout. In order to solve this problem, we should use a different similarity methodology that allows to give more importance to layout similarity. We are looking for alternative solutions to calculate similarity, as explained in *Section 7*.

The second cause of re-execution failure has been more difficult to discover than the first one just described. In this case the failure is due to the presence of a particular View object in the original sample's layout that however is

Figure 4.12: Example of similar samples with different layouts.

not present in the layouts of similar applications.

We discovered this particular case during the execution of *Test1*: during the stimulation of the original sample, we clicked on a link embedded in a TextView object. Re-executing the test on a similar application, the content of the TextView changed with the following disappearance of the link: hence, clicking on the TextView in the original sample led to open a new window, while the same click on the similar application did not generate any transition, making the re-execution test fail. This case is shown in **Figure 4.13**.

In *Test2* we tried to stimulate the same application, `com.keji.danti.207`, avoiding to click on that link: as expected, the re-execution has reached a score of 100%. Clearly, this particular failure case cannot be easily avoided, since a deep knowledge of the layout of the original and of the similar applications would be necessary.

Conclusions The results of the experiments supported our key intuition on the re-execution of UI stimulation traces: we demonstrated that if (malicious) behaviors are exercised during a manual test, it is quite likely that stimulating in the same way the UI of similar applications, similar behaviors are exercised. Moreover, these tests showed that exercising the UI of an application with the re-execution of stimulation trace allows to reach higher code coverage (i.e., exercising more behaviors) than automatic exercising strategies. We also discovered limitations in the re-execution technique adopted: one of them is tied to the current implementation of the similarity strategy used. As explained in *Section 7*, we can overcome this limitation adopting a different similarity approach. The second limitation is tied to the

Figure 4.13: Example of re-execution failure due to the presence of particular UI elements.

presence of unexpected UI elements in the layout of an application: at the moment there is not a solution to this problem. However, we observed that this limitation sporadically exhibits and, consequently, it does not affect the results of re-execution approach.

Chapter 5

Limitations

In this section we analyze the main limitations of PuppetDroid system. First of all we consider some implementation aspects that limit the performance of the system: we remind that PuppetDroid offers a highly interactive service, so reaching good performance during test execution is one of the main goals we sought during system development. Then, we comment on the problem of evasion, a common problem in the field of dynamic malware analysis. Finally, we recall the limitations related to the current re-execution implementation, which exhibited during experimental tests.

Performance

PuppetDroid provides an Android remote execution environment to perform tests: as explained in *Section 3.3.6*, users' devices communicate with our sandboxes through a VNC communication channel. Even if modern VNC encoding and compression algorithms allow to reach remarkable transmission performance, the overall speed of the session depends on network speed and computational power of both the endpoints. Considering the network, the bandwidth required to obtain good usability depends on the dimensions of the screen images exchanged between VNC client and server. We performed our tests using a smartphone with 320x480 pixels as screen size, labeled as *Normal* screen by Android, and we did not notice any performance degradation due to VNC transmission. However, using a smartphone with a 1280x800 pixel screen, labeled as *Extra Large* screen, we noticed a slowdown in the updating of the image. Indeed, this problem can be solved slightly modifying current PuppetDroid implementation in order to limit sandbox screen dimensions or introduce light image quality degradation in order to improve transmission speed.

Actually, the real performance bottleneck of PuppetDroid is the Android

emulator used as sandbox to host tests. As a matter of fact, Android emulator, or better QEMU, has to emulate ARM instructions on top of a X86 architecture provided by the host machine: this is clearly highly processor intensive and make the emulator run much slower than a real device. The first consequence of the poor performance of the emulator is that is very hard to run highly interactive applications, such as games.

Indeed, Google official documentation [16] suggests to exploit hardware acceleration and switch to the x86 version of Android emulator in order to make it run faster: we discarded this option because we prefer to execute our dynamic malware analysis in an environment as similar as possible to the physical environment used by real Android devices. Moreover, supporting ARM is currently necessary since there are Android malware samples that exploit ARM specific vulnerabilities to perform malicious actions.

Another alternative is to use physical devices instead of emulator: this solution can lead to a great performance improvement and, in addition, we obtain an optimal test environment. The drawback of this approach is that we have to modify the way we perform dynamic analysis: as a matter of fact, CopperDroid relies on the instrumentation of QEMU to perform system call tracking, while the use of a physical device would require the adoption of an instrumented Android image, such as done in solutions like TaintDroid or DroidBox.

Evasion

PuppetDroid executes test sessions in an emulated environment: this fact does not only imply performance limitations but it has been demonstrated [29] that a generic application can detect that it is running inside an Android emulator. As a matter of fact, the binary translation technique used in a QEMU fully-emulated environment does not properly respect the typical unpredictable behavior of a multitasking implementation based on timer interrupts: this is because QEMU, in order to speedup emulator performance, schedules new threads only after the execution of a complete basic block, while in a real environment the execution flow can be interrupted in any time inside a basic block. This is only an example about how to exploit particular implementation features of an emulator for evasion purposes: the point is that there always is the possibility that a malware can detect to be running in an emulated environment and can thus decide to hide its malicious behavior in order to not be analyzed.

A possible solution could be to use a physical device running an instrumented Android image. However, also in this case, since the analysis is performed inside the Android environment, if the malware succeeds in gaining root privileges, it can detect the presence of the analysis framework and hide

its malicious behavior.

Finding a solution to the evasion problem is still an open problem, not only for Android framework, but in general for the research field of dynamic malware analysis [24].

Re-execution

We recall here two limitations we found during the experimental evaluation of PuppetDroid re-execution approach. The first one is tied to the current solution used to calculate the similarity between two Android applications: it may happen that androsim considers two applications as similar, even if they greatly differ in the layout. This is actually a temporary limitation: in fact we are going to adopt a more reliable and scalable solution to calculate app similarity, as explained in *Section 7*. The second limitation is tied to the presence of a particular UI element in the original sample's layout that however is not present in the layouts of similar applications. At the moment we have not a solution to this particular case. Fortunately, we observed that it is a sporadic scenario and it does not affect the results of our re-execution approach.

Chapter 6

Related work

In this chapter we present an overview of relevant works in the fields related to this thesis: static malware analysis (*Section 6.1*) and Android applications similarity (*Section 6.2*).

6.1 Static malware analysis

As mentioned in *Section 2.2.1*, there are two approaches commonly used by analysts to study the behavior of an unknown program: static and dynamic analysis. We already introduced these two techniques and we discussed about main advantages and limitations of each. We present here the most relevant static analysis frameworks developed for Android.

The most common static analysis approach, at the base of anti-virus software products, is the filtering of samples basing on the research of sections of code that follows known malicious pattern of execution, named *signatures*. The main advantage of this approach is that it allows to develop thin client analyzers (i.e., anti-virus products), since it does not require much computational power. The clear drawback is that it is a blacklist approach, so malware signatures must be known in advance and anti-virus products must be constantly kept updated.

A lot of alternative static analysis approaches have been presented in the last years: we want to cite here DroidRanger [52], DroidMOSS [46] and RiskRanker [19].

DroidRanger is an interesting static analysis framework that leverages a fast signature-based scanning to detect known malware, in combination with a heuristic-based approach in order to identify zero-day threats. It has been developed to analyze the health of existing Android markets, having thus as

first requirement the ability to quickly analyze a huge amount of APK samples. In particular, in order to identify known malware samples, it retrieves from the manifest the information about requested permissions and if they match a known malware pattern of permission usage, then the sample is further analyzed using a signature based scanning. On the other hand, to detect unknown threats, the authors, basing on the study of known malware, defined some heuristics that allow to detect untrusted code in an application: these heuristics have been used to perform a first filtering of the samples to be analyzed. Then, the samples that passed this first step (i.e., that have been labeled as potentially malicious from one of the heuristics) are manually analyzed in order to establish if they are a zero-day threats or variants of existing malware. **Figure 6.1** presents an overview of DroidRanger architecture.

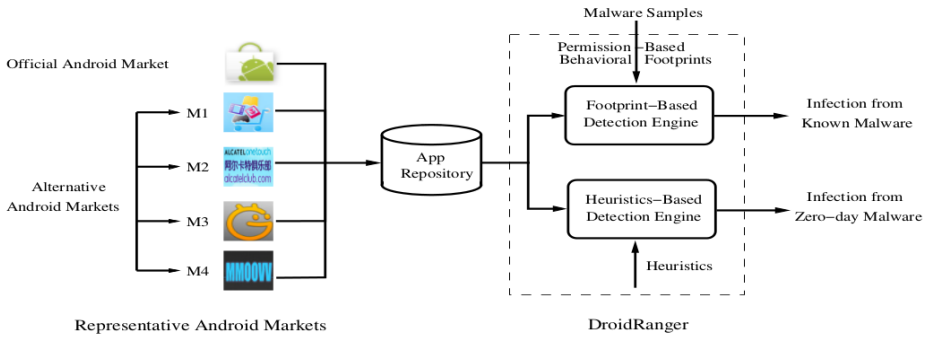

Figure 6.1: DroidRanger architecture.

This method has proved to be very effective, since it allowed to quickly analyze more than 200,000 applications and detect more than 200 malware samples, with 20% of them being zero-day threats.

DroidMOSS is an application similarity measurement system designed to detect repackaged APKs in third-party Android markets. The original idea at the base of this work is the use of a fuzzy hashing technique to generate a fingerprint of the bytecode of an application: the use of fuzzy hashing allows to obtain slightly different fingerprints for slightly different bytecodes. In this way it is possible to easily detect repackaged applications. The analysis is performed in 3 steps: first of all, each sample is analyzed in order to collect a set of identifying information. Then, fuzzy hashing is applied in order to generate a fingerprint from the list of bytecode instructions. Finally, the fingerprints of the application on alternative markets are compared with the fingerprints of the application on official Google Play market: if two applications have a high similarity score and are signed by two different authors, a repackaged application has been found. An overview of DroidMOSS architecture is shown in**Figure 6.2**.

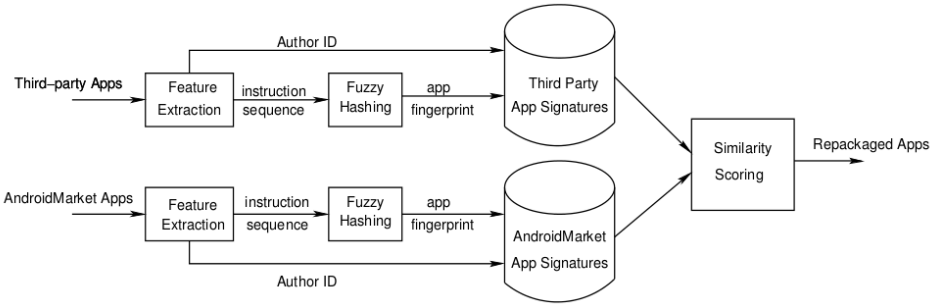

Figure 6.2: DroidMOSS architecture.

RiskRanker is an automated system that has been developed to detect zero-day malware that can compromise phone integrity, cause users financial loss or disclose sensitive information. We cite this work because it proposes an interesting approach to face code obfuscation with static analysis. As a matter of fact, RiskRanker uses two different types of analysis depending on the case a sample includes obfuscated code or not. In the case code obfuscation is not present, a two steps analysis is performed: the first step leverages a signature-based scanning to detect known exploits that allow to gain root privileges. The second step performs a control-flow analysis to identify critical paths to methods that can cause sensitive information disclosure or financial loss, without requesting user's approval. On the other hand, in case of code obfuscation, after a pre-processing phase to filter only applications using cryptographic methods, RiskRanker tries to identify applications that execute encrypted native code. To do that, it looks for execution paths that include both methods used to decrypt the contents of `res` or `asset` folders and methods used to execute native code through JNI.

6.2 Android application similarity

In this section we introduce some relevant works that address Android application similarity, a very useful information if you want to identify repackaged applications.

To our knowledge, the only currently available tool to calculate Android application similarity is androsim [5], distributed with Androguard reverse-engineering suite. This tool has been developed with the aim to easily discover pirated version of a genuine application or to evaluate the efficiency of code obfuscators. The authors leverage *Normalized Compressed Distance* (NCD) to approximate Kolmogorov complexity and to calculate the distance between two elements using real world compressors. In particular, given a compressor C, the NCD of two elements A and B is defined

by:

$$d_{NCD}(A,B) = \frac{L_{A|B} - min(L_A, L_B)}{max(L_A, L_B)}$$

where L is the length of a string, $L_A = L(C(A))$ and A/B is the concatenation of A and B.

The compressor C must be normal, i.e it has to satisfy the 4 inequalities:

1. **Idempotency**: C(xx) = C(x), and C(ϵ) = 0, where ϵ is the empty string.
2. **Monotonicity**: C(xy) $\geq$ C(x).
3. **Symmetry**: C(xy) = C(yx).
4. **Distributivity**: C(xy) + C(z) $\leq$ C(xz) + C(yz).

Moreover the compressor must be able to calculate C(x) within an acceptable amount of time.

The algorithm used to calculate the similarity between two applications works as follows:

- Extract the lists of methods from the bytecode of the two samples.
- Identify identical methods using an hashing comparison.
- Generate signatures for remaining methods.
- Identify similar methods using NCD.

So the global idea is to associate each method of the first application with others of the second application, excluding identical methods, by using NCD with an appropriate compressor, as shown in **Figure 6.3**. Finally, according to the number of identical methods and the distance between the remaining ones, a similarity score is given as output.

The algorithm just illustrated is very efficient in calculating the similarity between two applications: however pair-wise comparison does not scale if you have to calculate similarity on a large amount of applications. Juxtapp [20] faces this problem proposing a fast and scalable infrastructure for code similarity analysis among Android applications. To efficiently tackle the problem at large-scale, Juxtapp uses k-grams of opcode sequences of compiled applications and feature hashing. The authors used a combination of these techniques in order to have a robust and efficient representation of applications. Using this representation, this tool is able to quickly compute pair-wise similarity between applications in order to detect code reuse among hundreds of

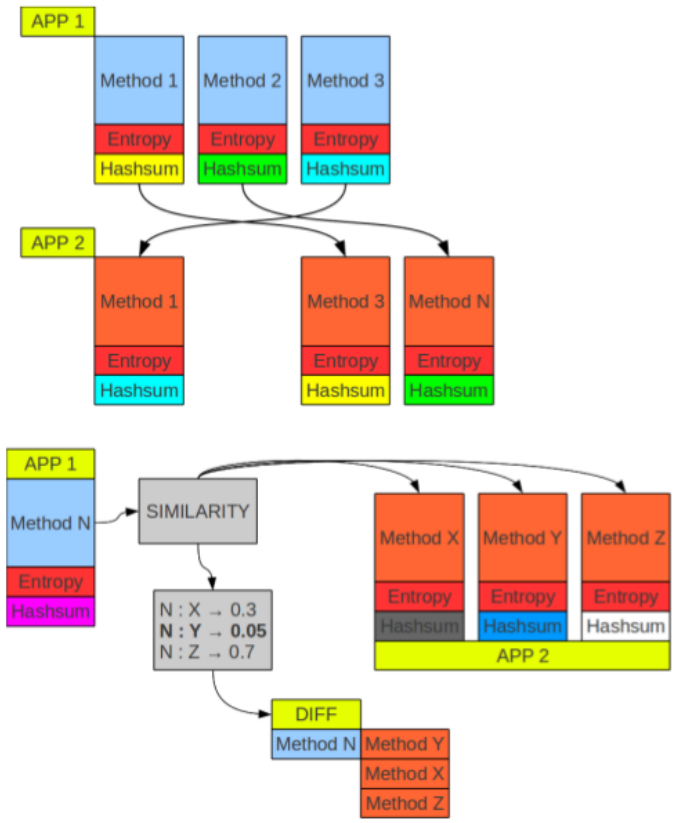

Figure 6.3: androsim analysis of similar methods.

thousands of applications. **Figure 6.4** shows the Juxtapp's workflow.

Another effective approach used to identify repackaged Android applications is the technique used by DroidMOSS to generate a fingerprint of the bytecode of an application leveraging fuzzy hashing (we mentioned this work in *Section 6.1*). Unfortunately this method presents the same scalability problem of androsim due to pair-wise comparison. In order to face this limitation, the same authors proposed PyggyApp [47]: this tool has been designed to quickly detect piggybacked samples (i.e., a special kind of repackaged applications that involve only the injection of additional code without modifying the already existing one). In particular, the authors propose a module decoupling technique that allows to partition the code of an application in primary and non-primary modules. Then, relying on the assumption that piggybacked applications share the same primary modules of the original sample, they extract from the primary module various semantic features and convert them into a feature vector. Finally, these feature vectors are organized into a metric space and a linearithmic search algorithm is used to quickly detect piggybacked applications. The architecture of PyggyApp is

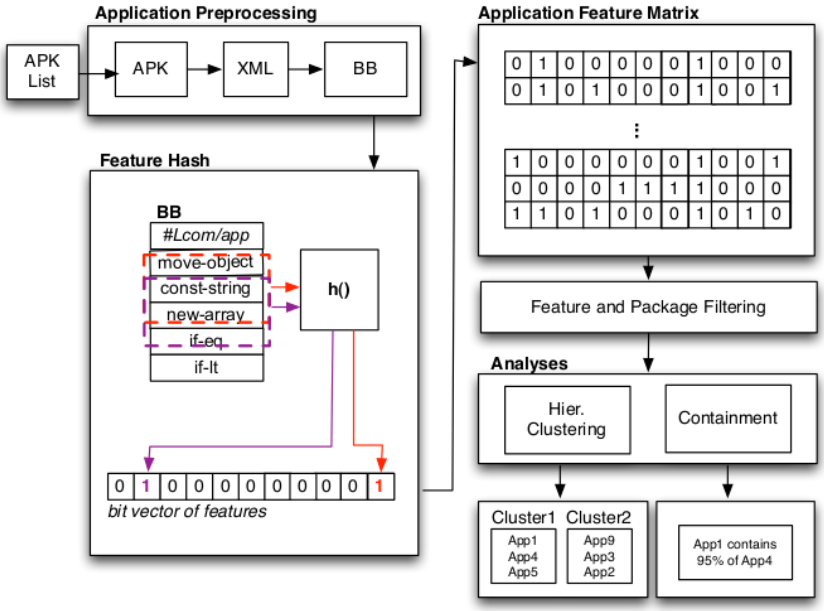

Figure 6.4: Juxtapp workflow.

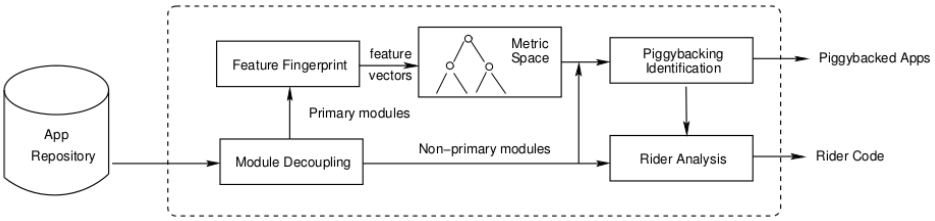

Figure 6.5: PyggyApp architecture.

presented in **Figure 6.5**.

Chapter 7

Conclusion and future work

The goal of our work was to propose a new approach to exercise the UI of Android applications in order to change the way malware analysis experiments are currently conducted and effectively stimulate potentially malicious behaviors. Our key intuition has been to analyze and reproduce the typical UI-interaction of a potential victim of the malware, stimulating in this way relevant behaviors for analysis purposes. To this end we designed and developed PuppetDroid, an Android sandboxed environment able to support both manual application testing, in order to collect new stimulation traces, and automatic application exercising, which leverages previously recorded UI stimulation traces.

In order to build PuppetDroid, we dealt with various challenges. First, we faced the problem of developing an efficient communication channel between users' Android devices and PuppetDroid sandboxes: for this, we leveraged VNC technology and implemented a thin Android VNC client and a custom VNC server. This solution allows our system to leverage human tester to manually exercise unknown applications and generate new stimulation traces. We then developed an original re-running technique that allows to re-execute the collected UI stimulation traces on applications with a layout similar to the one originally tested. We implemented this solution in Puppet ReRunner.

The combination of stimulation traces re-execution and collection of new traces from human tester makes our approach scale. We believe in fact that our system can attract the interest not only of security analysts but also of normal users that want to safely try potentially malicious applications on their devices. As last resort, our approach allows to leverage the abundance of available crowd-sourcing services in order to retrieve human workers to generate new stimulation traces.

We designed PuppetDroid to support different Android sandbox implementations in order to be flexible to future improvements. To this end, we defined a generic class implementing the concept of *Android Virtual Device*, which can be easily extended to support a specific sandbox for dynamic malware analysis: at the moment our system supports CopperDroid and DroidBox sandboxes.

Our experimental evaluations revealed that PuppetDroid human-driven stimulation is more effective than other approaches based on automatic UI stimulation: as a matter of fact, PuppetDroid is able to stimulate 254% of total behaviors and 200% of distinct behaviors more than the automatic stimulation methodologies, and, moreover, to stimulate 375% unique behaviors more respect than Monkey and 235% more respect than CopperDroid.

We experimentally demonstrated that our Puppet ReRunner tool is able to stimulate the same malicious behaviors seen during the original test also on similar applications. Moreover, our re-executed stimulation revealed to be more effective than automatic stimulation approaches: as a matter of fact, during our re-executed tests, we were able to stimulate 187% of total behaviors and 137% of distinct behaviors more than the automatic stimulation methodologies and 276% unique behaviors more respect than Monkey and 312% more respect than CopperDroid.

Our main contributions to the state of art have been the definition of a new approach to Android dynamic malware analysis that leverages human-driven UI stimulation to increase code coverage and the development of an original method to automatically exercise an unknown application re-using UI stimulation traces obtained from previously analyzed applications. Moreover, we implemented our approach in an easy-to-use service that leverages remote sandboxing to allow users to safely test potentially malicious applications without any risk of infection or information leakage.

Future Work

We plan to release PuppetDroid as a public service so as to leverage crowdsouring and collect new stimulation traces. To this end, first of all, we have to improve our web application in order to integrate user authentication and provide a better presentation of test results. As a matter of fact, the current available version is quite simple because it has been developed only for testing purposes.

As mentioned in *Section 5*, we are planning to substitute emulated sandboxes with physical devices. This solution will allow to improve overall performance giving the possibility to test also highly interactive applications. Another

advantage of using physical device is obtaining an analysis system more resilient to evasion. Moreover, regarding this aspect, we want to follow the example of AppsPlayground [36] and enrich our test environment with realistic data, such as contacts, SMS, pictures, files on SD card and so on: in this way we can provide a test environment that looks like a real device, preventing possible detection capabilities of modern malware.

In *Section 3.3.7.2* we showed how our current similarity solution, based on the use of androsim, revealed to be too slow for our purposes. Then, we are looking for alternative solutions for the computation of similarity between Android applications: since our aim is to quickly detect repackaged applications, a possible solution is to integrate in PuppetDroid one of the scalable approaches presented in Juxtapp [20] and PyggyApp [47].

Appendix A

Android tools

PuppetDroid is essentially focused on creating an Android execution environment to test Android applications. Therefore, we introduce here some of the Android SDK tools that are used in PuppetDroid implementation and that are critical to the operation of the test environment.

A.1 Android Emulator

One of the most important development tools included in the Android SDK is definitely the handset emulator. Based on QEMU, it can natively emulate all the official Android ROMs, mimicking all of the hardware and software features of a typical mobile device.

The emulator utilizes the *Android Virtual Device* (AVD) configurations to let users define certain hardware aspects of the emulated environment to easily test different Android platforms and hardware permutations. Since PuppetDroid has the goal to create a test environment that is as similar as possible to the real environment provided by the user device, it is important to understand how to correctly customize an AVD.

Creating an AVD can be done either by using the graphical interface (AVD manager) or by leveraging the command line tool `android create avd`, whose syntax is reported in **Listing A.1**.

Listing A.1: AVD creation from command line

```
$ android create avd -n <name> -t <targetID> -b <abi> -s <skin>
    [-<option> <value>] ...
```

The parameter named `targetID` is the identifier of the Android platform the

new AVD will be based on. The list of all the supported platforms and their respective identifiers can be obtained by issuing the command `android list targets`. The parameter `abi` indicates the *Application Binary Interface* (ABI), i.e., the low-level interface used to communicate with the operating system that the emulator instance will load. The two ABI currently supported in Android are *armeabi* and *armeabi-v7a*. The parameter `skin` indicates the screen dimensions, in pixel, of the emulator. All these parameters must be correctly set to match the features of user's device.

Once an AVD is defined, it can be launched by using the command `emulator` followed by the AVD name (**Listing A.2**).

Listing A.2: AVD launching from command line

```
$ emulator -avd <avd_name> [<options>]
```

This command takes various parameters. For example, launching the emulator with the `-no-window` parameter will start the emulator in headless mode, allowing its execution without a *Graphical User Interface* (GUI).

One of the most criticized aspects of the Android emulator is its slow startup. Indeed, starting the emulator can really take a long time, ranging from 2 to 10 minutes, depending on the hardware it is hosted on. To reduce this time, starting from the Android SDK R9, Google introduced the possibility to leverage the *snapshots* functionality that allows to start the emulator skipping all the time-consuming Android startup procedure. Since in PuppetDroid each test execution, in order to fit the wide range of available Android devices, requires an emulator with an unpredictable set of features, this functionality cannot be exploited: in fact, it would require a snapshot for each possible screen configuration. However, to partially mitigate this problem, we preload a clean image of the user data before the emulator is started: in this way we can reduce the startup time because we cut the time required to load all the default user applications.

In order to run appropriately, the Android emulator needs various image files (**Table A.1**). In particular, in order to correctly record UI stimulation and perform dynamic malware analysis, PuppetDroid system uses properly modified system images that are loaded during emulator startup.

More information on how to use the Android emulator can been retrieved from Google official documentation [16, 14].

Filename	Description
`kernel-qemu`	The emulator-specific Linux kernel image
`ramdisk.img`	The ramdisk image used to boot the system
`system.img`	The initial system image
`userdata.img`	The initial data partition image
`userdata-qemu.img`	An optional persistent data partition image
`system-qemu.img`	An optional persistent system image
`cache.img`	An optional cache partition image
`sdcard.img`	An optional SD Card partition image
`snapshots.img`	An optional state snapshots image

Table A.1: Required and optional emulator image files. The list is obtained by using the command `emulator -help-disk-images`.

A.2 Android Debug Bridge

Android Debug Bridge (ADB) is a command line tool used to communicate with an emulator instance or connected Android device. It is a client-server program that comprises three components:

Client – It runs on the user machine and is used to issue commands to the Android system.

Server – It runs as a background process on the user machine. It manages communication between the client and the ADB daemon running on an emulator or device.

Daemon – It runs as a background process on each emulator or device and executes the commands received from the ADB clients.

The ADB client application can be accessed through the `adb` command, whose syntax is shown in **Listing A.3**.

Listing A.3: `adb` **command syntax**

```
$ adb [-d|-e|-s <serialNumber>] <command>
```

The full list of supported options and subcommands can be found in the Android official documentation [13]. We hereby report only the main commands we leveraged in PuppetDroid. These commands are summarized and described in **Table A.2**.

Command	Description
`install <path-to-apk>`	Push and install an Android application (specified as a full path to an APK file) to an emulator/device.
`forward <local> <remote>`	Forward socket connections from a specified local port to a specified remote port on the emulator/device instance.
`logcat [option] [filter-specs]`	Print log data to the screen.
`shell <command>`	Run a remote shell command on the emulator/device.
`push <local> <remote>`	Copy a file to the emulator/device.
`pull <remote> <local>`	Copy a file from the emulator/device.

Table A.2: ADB main commands used in PuppetDroid implementation.

A.3 Monkey

As part of the functional-level application testing tools, the Android SDK includes a tool called *UI/Application Exerciser Monkey* (usually referred to as monkey). Monkey runs on the emulator or device and generates pseudo-random streams of user events such as clicks, touches, or gestures. It is mainly used to stress-test applications and includes some non-documented features to replicate a specific sequence of inputs.

Monkey is usually used by by some of the Android sandboxes that perform automatic dynamic malware analysis in order to stimulate potentially malicious behaviors. It has been used in our experimental tests to compare its pseudo-random stimulation with PuppetDroid human-driven stimulation. Results of this comparison can be found in *Section 4.1*.

Monkey can be run by invoking the `monkey` command from an Android shell (accessible via adb). In particular, the typical command used to stress-test an application is reported in **Listing A.4**.

Listing A.4: Monkey command used to stress-test an application

```
$ adb shell monkey [options] <event-count>
```

Where the parameter `event-count` indicates the number of pseudo-random events that must be generated by the tools.

More information on Monkey tool can be found in [17].

A.4 HierarchyViewer and Viewserver

One of the most interesting features of PuppetDroid is the possibility to re-execute the human-driven stimulation used in a test to stimulate other applications that have a layout similar to the one of the originally tested application. In order to reach this goal, knowing which point on the screen has been touched by the user is not enough: we need to exactly know which element, inside the Android application layout, has been stimulated in order to achieve a robust re-execution.

Before going any further, we want to spend a few words about Android *User Interface* (UI) Layout. All UI elements in an Android application are built using *View* and *ViewGroup* objects. A *View* is an object that draws something on the screen that the user can interact with. A *ViewGroup* is an object that holds other *View* (and *ViewGroup*) objects in order to define the layout of the interface. The user interface for each component of an Android application is defined using a hierarchy of *View* and *ViewGroup* objects, as shown in **Figure A.1**. Each view group is an invisible container that organizes child views, while the child views may be input controls or other widgets that draw some part of the UI. A detailed description of Android UI Layout can be found in Google official documentation [15].

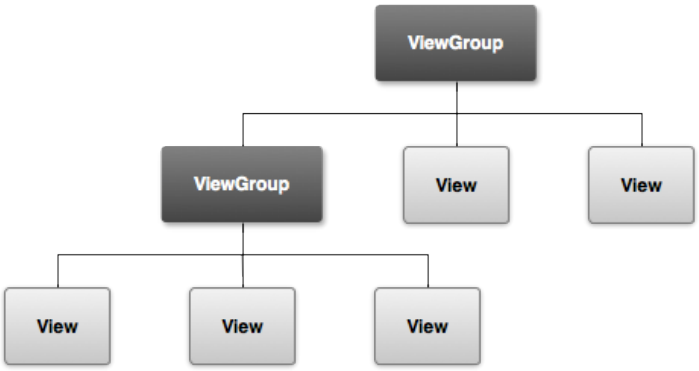

Figure A.1: Illustration of a view hierarchy, which defines a UI layout.

Since view elements can be added at runtime by an application, we need a way that allows us to dynamically extract the exact view hierarchy whenever the user interacts with the UI. In order to do so, we exploit another interesting tool provided by the Android SDK: the HierarchyViewer, a visual tool that can be used to inspect the application user interfaces to analyze

all the aspects of an applications layout at runtime. A screenshot of the HierarchyViewer tool execution is reported in **Figure A.2**.

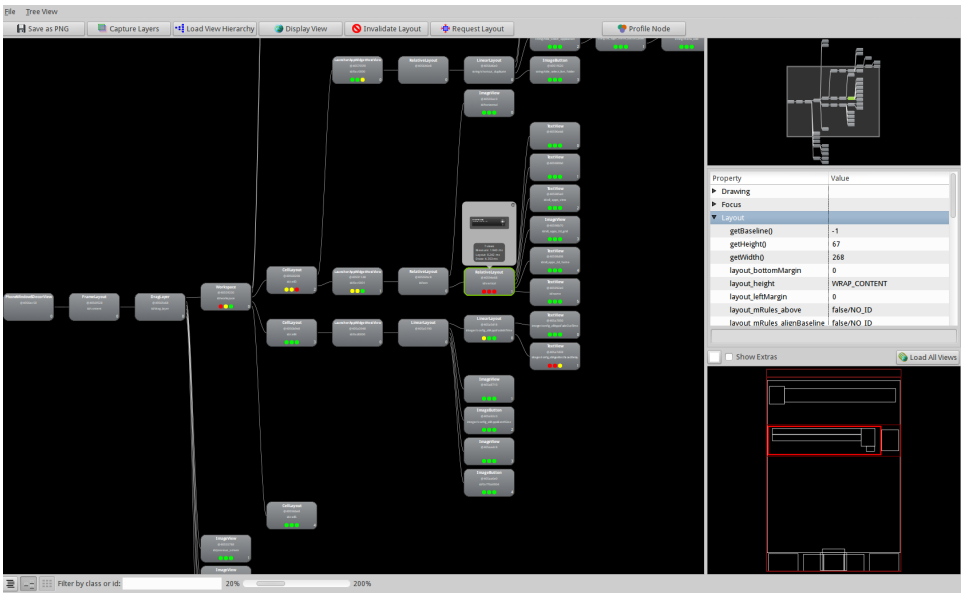

Figure A.2: Screenshot of the HierarchyViewer interface.

As we can see, the tool shows a graphical hierarchy of the views displayed on the running Android instance. As in the case of other SDK tools, HierarchyViewer lacks of official documentation. Indeed, there is no explanation on how the tool internally works and no mention about the origin of the shown data. Nevertheless, analyzing HierarchyViewer source code, we found out that it retrieves the displayed information from an Android internal component called ViewServer.

When started, ViewServer opens a socket on a specified local port to accept commands from a client (usually HierarchyViewer) to dump the current view state of the device. The ViewServer dispatches these calls by serializing the view state and transmitting it to the client over the socket.

The ViewServer service can be launched by invoking the `service` command in an Android shell, with the syntax reported in **Listing A.5**.

Listing A.5: `service` command syntax

```
root@android:/ # service call <service_name> <service_code> \
>  [i32 <int_value> | s16 <str_value>]
```

The parameters' meaning is:

- `service_name` – the name of the service to communicate with. In the case of ViewServer it must be `window`.

- `service_code` – an integer code identifying the action to be performed by the service. In the case of ViewServer, the acceptable values are:
  - `1` = starts service
  - `2` = stops service
  - `3` = checks service status
- `int_value` and `str_value` – an integer or string parameter that is passed as argument to the called service. In the case of ViewServer the expected value is an integer number representing the socket port where the service will start listening for connection.

Thus, for instance, starting the ViewServer service to listen on the socket port 4939 can be done with the command reported in **Listing A.6**.

Listing A.6: ViewServer start command

```
root@android:/ # service call window 1 i32 4939
Result: Parcel(00000000 00000001    '........')
```

When correctly executed, the command returns:

```
Result: Parcel(00000000 00000001    '........')
```

otherwise the displayed string will be:

```
Result: Parcel(00000000 00000000    '........')
```

We derived a list of accepted commands (**Table A.3**) from ViewServer's source code.

Command	Description
`LIST`	Dumps the list of all the *Activity* currently running on the device, along with their hashcodes.
`GET_FOCUS`	Gets the name and hashcode of the focused *Activity*.
`DUMP root_hashcode`	Dumps the currently displayed view hierarchy, starting from the given root view. If `root_hashcode = -1`, then all the views are dumped.

Table A.3: ViewServer commands reference.

The data returned by the `LIST` command is basically a list of "`<hashcode> <activity>`" pairs (separated by newlines), while the `GET_FOCUS` commands returns just a single "`<hashcode> <activity>`" pair (**Listing A.7 and A.8**).

Listing A.7: Data returned by ViewServer LIST command

```
LIST
```

```
40517538 com.android.internal.service.wallpaper.ImageWallpaper
40917368 com.android.launcher/com.android.launcher2.Launcher
40935528 TrackingView
40a28018 StatusBarExpanded
407af408 StatusBar
DONE.
```

Listing A.8: Data returned by ViewServer GET_FOCUS command

```
GET_FOCUS
40917368 com.android.launcher/com.android.launcher2.Launcher
```

The data returned by the DUMP command is more complex. It basically contains the serialization of all the information regarding the views displayed on the screen, starting from a given root view. This data follows the format we report in **Listing A.9** and can be used to programmatically build a local hierarchy of the views shown on a running Android instance.

Listing A.9: Format of data returned by ViewServer DUMP command

```
<tree_node_depth><component_name>@<hashcode> [<property_name>=<
    value_length,property_value>] ...
...
```

<tree_node_depth> is expressed as a sequence of whitespaces, denoting the depth of the node with respect to the root node.

It is important to note that the DUMP command appears to be quite slow. One entire view hierarchy DUMP can take up to 40 seconds. This delay is due to Android's ViewServer component that makes heavy use of Java introspection to discover which members are to be dumped. We explain how we faced this problem in *Section 3.3.7*.

Bibliography

[1] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp Styp-Rekowsky. AppGuard - Real-time policy enforcement for third-party applications. Technical report, Saarland University, 2012. URL http://scidok.sulb.uni-saarland.de/volltexte/2012/4902.

[2] Mario Ballano. Android malware. In *ITU Regional forum on Cybersecurity*. Symantec, 2012.

[3] Scott Bicheno. Global Smartphone Installed Base Forecast by Operating System for 88 Countries: 2007 to 2017. http://www.strategyanalytics.com/default.aspx?mod=reportabstractviewer&a0=7834, October 2012.

[4] Canalys. Smart phones overtake client PCs in 2011. http://www.canalys.com/newsroom/smart-phones-overtake-client-pcs-2011, February 2012.

[5] Anthony Desnos and Geoffroy Gueguen. Android: From Reversing to Decompilation. http://developer.android.com/tools/help/adb.html.

[6] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1924943.1924971.

[7] ESET Latin America Lab. Trends for 2013, Astounding growth of mobile malware. Technical report, ESET Latin America Lab, November 2012.

[8] F-Secure. Mobile Threat Report Q4 2012. Technical report, F-Secure, March 2013.

[9] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0948-6. doi: 10.1145/2046707.2046779. URL http://doi.acm.org/10.1145/2046707.2046779.

[10] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '11, pages 3–14, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1000-0. doi: 10.1145/2046614.2046618. URL http://doi.acm.org/10.1145/2046614.2046618.

[11] Girault, Emilien. Reversing Google Play and Micro-Protobuf applications. http://www.segmentationfault.fr/publications/reversing-google-play-and-micro-protobuf-applications/.

[12] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. Reran: timing- and touch-sensitive record and replay for android. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 72–81, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3. URL http://dl.acm.org/citation.cfm?id=2486788.2486799.

[13] Google Inc. Android Debug Bridge. http://developer.android.com/tools/help/adb.html, .

[14] Google Inc. Managing AVDs from the Command Line. http://developer.android.com/tools/devices/managing-avds-cmdline.html, .

[15] Google Inc. Android User Interface. http://developer.android.com/guide/topics/ui/index.html, .

[16] Google Inc. Using the Emulator. http://developer.android.com/tools/devices/emulator.html, .

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

[18] Michael Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic detection of capability leaks in stock Android smartphones. In *Proceedings of the 19th Network and Distributed System Security Symposium*, NDSS'12, February 2012. URL http://www.csc.ncsu.edu/faculty/jiang/pubs/NDSS12_WOODPECKER.pdf.

[19] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. Riskranker: scalable and accurate zero-day android mal-

ware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, MobiSys '12, pages 281–294, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1301-8. doi: 10.1145/2307636.2307663. URL http://doi.acm.org/10.1145/2307636.2307663.

[20] Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song. Juxtapp: a scalable system for detecting code reuse among android applications. In *Proceedings of the 9th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA'12, pages 62–81, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 978-3-642-37299-5. doi: 10.1007/978-3-642-37300-8_4. URL http://dx.doi.org/10.1007/978-3-642-37300-8_4.

[21] Oliva Hou. A Look at Google Bouncer. http://blog.trendmicro.com/trendlabs-security-intelligence/a-look-at-google-bouncer/, July 2012.

[22] IDC. Apple Cedes Market Share in Smartphone Operating System Market as Android Surges and Windows Phone Gains, According to IDC. http://www.businesswire.com/news/home/20130807005280/en/Apple-Cedes-Market-Share-Smartphone-Operating-System, August 2013.

[23] Google Inc. Android open source project: Philosophy and goals. http://source.android.com/about/philosophy.html.

[24] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. Barebox: efficient malware analysis on bare-metal. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 403–412, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0672-0. doi: 10.1145/2076732.2076790. URL http://doi.acm.org/10.1145/2076732.2076790.

[25] International Secure Systems Lab. Andrubis: A tool for analyzing unknown android applications. http://blog.iseclab.org/2012/06/04/andrubis-a-tool-for-analyzing-unknown-android-applications-2/.

[26] Hiroshi Lockheimer. Android and Security. http://googlemobile.blogspot.it/2012/02/android-and-security.html, February 2012.

[27] Lookout Mobile Security. Lookout Mobile Security: State of Mobile Security 2012. Technical report, Lookout Mobile Security, September 2012.

[28] Aravind MacHiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 ACM*

Symposium on Foundations of Software Engineering, FSE'13, 2013.

[29] Matenaar, Felix and Schulz, Patrick. Detecting Android Sandboxes. http://www.dexlabs.org/blog/btdetect.

[30] McAfee. McAfee Threats Report: Second Quarter 2013. Technical report, McAfee, 2013.

[31] McAfee Inc. Virus Profile: Android/BaseBridge.G. http://home.mcafee.com/virusinfo/virusprofile.aspx?key=665341, .

[32] McAfee Inc. 'FakeInstaller' Leads the Attack on Android Phones. http://blogs.mcafee.com/mcafee-labs/fakeinstaller-leads-the-attack-on-android-phones, .

[33] Donald Melanson. Eric Schmidt: Google now at 1.5 million Android activations per day. http://techcrunch.com/2012/09/05/eric-schmidt-there-are-now-1-3-million-android-device-activations-per-day/, April 2013.

[34] Mobile Antivirus. New Android Trojan Detected, Called BaseBridge. http://www.mobiantivirus.org/antivirus/basebridge.html.

[35] Jon Oberheide and Charlie Miller. Dissecting the Android Bouncer. In *SummerCon 2012*, June 2012. URL http://jon.oberheide.org/blog/2012/06/21/dissecting-the-android-bouncer/.

[36] Vaibhav Rastogi, Yan Chen, and William Enck. Appsplayground: automatic security analysis of smartphone applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, CODASPY '13, pages 209–220, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1890-7. doi: 10.1145/2435349.2435379. URL http://doi.acm.org/10.1145/2435349.2435379.

[37] Alessandro Reina, Aristide Fattori, and Lorenzo Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In *Proceedings of the 6th European Workshop on System Security*, EUROSEC'13, April 2013.

[38] Tristan Richardson. The rfb protocol - version 3.8. Technical report, RealVNC Ltd, 2010.

[39] Dave Smith. Mastering the android touch system. In *Proceedings of the 2012 Fourth Android Developer Conference*, AnDevConIV, 2012.

[40] Sophos. Security Threat Report 2013. Technical report, Sophos, 2013.

[41] Symantec Corporation. Android.Basebridge. http://www.symantec.com/security_response/writeup.jsp?docid=2011-060915-4938-99.

[42] Trend Micro. Repeating History. Technical report, Trend Micro, January 2013.

[43] Chris Welch. Google: Android app downloads have crossed 50 billion, over 1M apps in Play. `http://www.theverge.com/2013/7/24/4553010/google-50-billion-android-app-downloads-1m-apps-available`, July 2013.

[44] Lok Kwong Yan and Heng Yin. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 29–29, Berkeley, CA, USA, 2012. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=2362793.2362822`.

[45] 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 *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '12, pages 93–104, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1666-8. doi: 10.1145/2381934.2381950. URL `http://doi.acm.org/10.1145/2381934.2381950`.

[46] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, CODASPY '12, pages 317–326, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1091-8. doi: 10.1145/2133601.2133640. URL `http://doi.acm.org/10.1145/2133601.2133640`.

[47] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. Fast, scalable detection of "piggybacked" mobile applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, CODASPY '13, pages 185–196, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1890-7. doi: 10.1145/2435349.2435377. URL `http://doi.acm.org/10.1145/2435349.2435377`.

[48] Yajin Zhou and Xuxian Jiang. Android malware genome project. `http://www.malgenomeproject.org/`.

[49] Yajin Zhou and Xuxian Jiang. An analysis of the anserverbot trojan, 2011.

[50] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 95–109, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4681-0 doi: 10.1109/SP.2012.16. URL `http://dx.doi.org/10.1109/SP.2012.16`.

[51] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent W. Freeh. Taming information-stealing smartphone applications (on android). In *Proceedings of the 4th international conference on Trust and trustworthy computing*, TRUST'11, pages 93–107, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-21598-8. URL `http://dl.acm.org/citation.cfm?id=2022245.2022255`.

[52] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium*, NDSS'12, February 2012.

Acronyms

ABI *Application Binary Interface*

ACG *Activity Call Graph*

ADB *Android Debug Bridge*

API *Application Programming Interface*

APK *Android Package*

AV *Anti-Virus*

AVD *Android Virtual Device*

CFG *Control Flow Graph*

CPU *Central Processing Unit*

DEX *Dalvik EXecutable*

DPI *Dots Per Inch*

DVM *Dalvik Virtual Machine*

ER *Entity-Relationship*

FCG *Function Call Graph*

GNU GPL *GNU General Public License*

GUI *Graphical User Interface*

I/O *Input/Output*

IP *Internet Protocol*

IPC *Inter-Process Communication*

ISA *Instruction Set Architecture*

JCE *Java Cryptographic Extension*

JNI *Java Native Interface*

NCD *Normalized Compressed Distance*

ORM *Object Relational Mapper*

OS *Operating System*

RFB *Remote Framebuffer*

RDBMS *Relational Database Management System*

RPC *Remote Procedure Call*

SDK *Software Development Kit*

SQL *Structured Query Language*

TCG *Tiny Code Generator*

TCP *Transmission Control Protocol*

TLS *Transport Layer Security*

UI *User Interface*

UML *Unified Modeling Language*

URL *Uniform Resource Locator*

UUID *Universally Unique Identifier*

VM *Virtual Machine*

VMI *Virtual Machine Introspection*

VNC *Virtual Network Computing*