

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



DROIDSAge: AN AUTOMATED
ANDROID SANDBOX GENERATOR

Relatore:

Prof. Stefano ZANERO

Correlatore:

Prof. Federico MAGGI

Tesi di Laurea Magistrale di:

Eros LEVER, matr. 766022

Anno Accademico 2012–2013

Abstract

Since its public release in 2008, the Android operating system had an explosive growth in market share. This fact, coupled with the openness with respect to third party applications made it the perfect target platform for mobile malware, that is malicious software targeting mobile platforms. Mobile malware is typically disguised as applications with hidden functions that aim to steal sensitive data from a user's device. Mobile devices are in fact rich in sensitive data, such as personal contacts, messages, emails and geographical coordinates that are very common to find on smartphones.

The ever-increasing number of applications that are published everyday make it challenging for security experts to protect Android users verifying the behaviour of unknown applications and identify new malware threats. In this scenario, the need of an automated analysis platform has always been present. Techniques for automated analysis are mainly divided in two categories, static and dynamic approaches. Static analysis is performed searching the application's code for known patterns that may correspond to malicious behaviours. Dynamic analysis is instead the process of executing the application and trace specific behaviours as they happen at runtime.

Currently existing solutions that allow tracing the behaviour of a generic application at runtime, thus offering a blackbox dynamic analysis environment, are still limited in terms of compatibility with Android versions. Some of these solutions furthermore employ deep modifications to the underlying system that narrow the target reach only to very specific device versions, requiring a new environment to be compiled and built from source.

Creating a new environment compiling it from the source code is not always possible, for instance, it is common for hardware manufacturer to only release prebuilt binaries without distributing the sources.

In this scenario we propose a system able to generate blackbox dynamic

analysis platforms for Android applications obtained altering existing environments rather than creating new ones. The modifications applied by the system target solely the Dalvik bytecode, allowing compatibility across multiple Android versions.

Behaviours are traced by means of instrumentation of Application Programming Interfaces (APIs) that are declared or invoked in the Dalvik bytecode present in the original environments. Instrumentation of the APIs is performed modifying the Dalvik bytecode and injecting instructions that will be executed at runtime when the target APIs are invoked. Those instructions have then to take actions such as logging each invocation.

The environments to alter are supplied as Android system images, which represent the disk image for the partition that held system specific applications and libraries. A system image may also contain optimised Dalvik bytecode. To instrument this variant of bytecode, it will first be converted to regular bytecode translating optimised instructions in their unoptimised equivalent. The resulting instrumented system image can then be used with the Android emulator or flashed on physical devices.

The developed system has been compared to existing state of the art solutions with a series of tests performed on a dataset of sample applications composed of both malware and benign applications.

The main contributions of this work are: an automated instrumentation of Dalvik bytecode and its optimised variant, compatibility with possibly every Android version including physical devices, highly configurable customisation of the generated analysis platform and avoiding breaking the signature verification despite the bytecode modification.

Sommario

Dal rilascio del primo dispositivo con sistema operativo Android ad oggi, questi dispositivi sono riusciti a raggiungere un'alta diffusione, spopolando come piattaforme mobili per smartphone e tablet. Questo fatto insieme all'apertura del sistema operativo verso applicazioni di terze parti, lo ha reso l'obiettivo perfetto per applicazioni malevole, note come *mobile malware*. L'elevatissimo numero di applicazioni rilasciate ogni giorno per questa piattaforma sta rendendo complicato il compito di proteggere l'utente del dispositivo da applicazioni malevole, volte ad esempio a rubare dati personali di cui questi dispositivi sono particolarmente ricchi.

Per gestire tali volumi di applicazioni, esperti di sicurezza informatica si devono necessariamente avvalere di tecniche di analisi automatizzata. Tale analisi deve essere in grado di esaminare applicazioni sconosciute, a questo fine non si prestano tecniche di analisi *white-box* (a scatola aperta) che necessitano di informazioni pregresse, tipicamente il codice sorgente dell'applicazione.

Gli approcci *black-box* (a scatola chiusa) disponibili si dividono tipicamente in tecniche statiche e dinamiche, anche se esistono approcci ibridi. Analisi statica è un processo che analizza il solo codice dell'applicazione senza di fatto eseguirlo, cercando di individuare schemi noti che potrebbero ricondurre a comportamenti malevoli da parte dell'applicazione. Le tecniche di analisi dinamica prevedono invece l'esecuzione dell'applicazione in un ambiente, sia esso reale o virtualizzato, dove specifici comportamenti vengono individuati durante l'esecuzione stessa.

Il contesto in cui questo lavoro si pone è la realizzazione un sistema *black-box* di analisi dinamica in grado di tracciare comportamenti come ad esempio l'invio di un SMS durante l'esecuzione di una o più applicazioni in un dispositivo Android, reale o emulato.

Le soluzioni ora disponibili in questo contesto sono molteplici, ma spesso

limitate a specifiche versioni del sistema operativo Android, o a specifici dispositivi. Inoltre, molti degli approcci presenti nello stato dell'arte richiedono la compilazione da sorgente del sistema Android. Questo può non essere sempre possibile, si veda d'esempio il fatto che è consuetudine per le case produttrici di componenti *hardware* non rilasciare il codice sorgente, ma solo i binari compilati. Questo lavoro si propone come sistema di generazione automatizzata di ambienti di analisi *black-box* che si avvalgano di tecniche di analisi dinamica per tracciare determinati comportamenti durante l'esecuzione di un'applicazione Android.

Il nostro sistema analizza intere immagini di sistema Android, andando ad instrumentarne il codice affinché un set personalizzabile di *Application Program Interface (API)* sia monitorato durante l'esecuzione. Comportamenti possono essere poi identificati da singole invocazioni di *API* o da gruppi di esse.

Le *API* da monitorare devono essere individuate dal sistema nel bytecode Dalvik, ovvero il codice che verrà eseguito dalla macchina virtuale Dalvik presente in Android. Questa macchina virtuale ha lo scopo di astrarre il codice da eseguire rispetto all'*hardware* presente sulla macchina. Per questo motivo, modificando il bytecode è possibile mantenere la compatibilità con potenzialmente tutte le versioni di Android.

E' possibile che una immagine di sistema contenga anche del bytecode Dalvik ottimizzato. Per instrumentare le *API* in questa variante del bytecode Dalvik è stato necessario introdurre al sistema un'ulteriore funzionalità, ovvero la conversione del bytecode ottimizzato nella sua versione non ottimizzata, andando a tradurre ogni istruzione ottimizzata nel suo equivalente. Il nostro sistema effettua una mappatura del codice in tutta l'immagine di sistema, creando un grafo delle classi ed un grafo delle invocazioni. Questi due grafi verranno poi utilizzati sia in fase di de-ottimizzazione, sia in fase di instrumentazione delle *API* da tracciare.

Il sistema da noi realizzato è stato poi utilizzato per generare un set di ambienti di analisi, che sono poi stati confrontati con soluzioni esistenti nello stato dell'arte. Abbiamo eseguito i confronti rispetto ad un set di applicazioni composto sia da applicazioni malevole, sia da applicazioni non malevole. Il confronto ha inoltre richiesto sia lo sviluppo di funzioni richiamate quando le *API* instrumentate vengono invocate, sia di un sistema di test automatico. Le funzioni realizzate hanno lo scopo di aggiungere qualora necessario al registro

di *log* il fatto che una determinata API sia stata invocata.

Il sistema di test automatico realizzato prevede l'avvio automatico di un set di emulatori Android, ciascuno dei quali esegue un'istanza di un ambiente instrumentato su cui viene installata ed eseguita un'applicazione. Per garantire l'uso di un sistema pulito, sono stati creati degli *snapshot*, ovvero delle istantanee di sistema da cui far partire un test.

Per rendere i test più realistici, si è cercato di riprodurre l'interazione di un utente con il sistema su cui viene eseguito il test. L'interazione generata è stata resa deterministica a scopo di ripetibilità, e composta sia da eventi *touch* sullo schermo, sia da tasti premuti.

I test effettuati hanno dimostrato che, seppur con differenze più o meno elevate dovute a variabili nell'ambiente sia interno che esterno, che gli ambienti instrumentati generati hanno un funzionamento simile a quello di sistemi presenti nello stato dell'arte.

Contents

Contents	vii
List of Figures	xi
List of Tables	xiii
1 Introduction	1
2 Background and motivation	5
2.1 Android malware	5
2.2 State of the art	6
2.2.1 TaintDroid	6
2.2.2 DroidBox	7
2.2.3 Andrubis	9
2.2.4 DroidScope	9
2.2.5 CopperDroid	10
2.2.6 API Monitor	11
2.2.7 Apps Playground	12
2.3 Goals and challenges	12
3 The Android runtime	15
3.1 The Dalvik Virtual Machine	15
3.2 Comparison with the Java Virtual Machine	16
3.3 Structure of a DEX file	18
3.4 Structure of an ODEX file	19
3.5 Signature verification in Android	22
4 Approach	25
4.1 System image generation	25

4.2	Application analysis	27
4.3	Instrumentation hooks	28
4.4	Comparison with other state of the art solutions	30
4.4.1	Comparison with API Monitor	30
4.4.2	Comparison with TaintDroid	31
4.4.3	Comparison with DroidBox	31
4.4.4	Comparison with Andrubis	32
5	Implementation	33
5.1	Bytecode injection	33
5.1.1	Reasons to inject bytecode	33
5.1.2	Injection technique	34
5.1.3	Bytecode adaptation	34
5.2	Bytecode instrumentation	35
5.2.1	Instrumentation points	36
5.2.2	Internal instrumentation of a method	37
5.2.3	External instrumentation of a method	37
5.3	Instrumentation hooks	38
5.3.1	The Payload class	38
5.3.2	Call stack inspection	39
5.3.3	Serialisation of objects to JSON	41
5.3.4	Logging of APIs	42
5.4	System image generation	44
5.4.1	Image extraction	46
5.4.2	Dalvik bytecode identification	46
5.4.3	Conversion of ODEX into DEX	47
5.4.4	Class graph and call graph generation	50
5.4.5	User decision of target APIs	52
5.4.6	Instrumentation	53
5.4.7	Bytecode optimisation	54
5.4.8	Image creation	54
6	Experimental validation	57
6.1	Testbed environments	58
6.1.1	System images	58
6.1.2	Stimulation of a single application	60
6.1.3	Automatic stimulation of multiple applications at once	62

6.2	Dataset	63
6.2.1	Normal applications	63
6.2.2	Malware applications	63
6.3	Test description	64
6.3.1	No stimulation	64
6.3.2	Stimulation with random events	65
6.4	Test results	66
6.4.1	Comparison of DroidBox and instrumented GingerBread	66
6.4.2	Comparison of DroidBox and GingerBread with GSF	67
6.4.3	Comparison of DroidBox and ICS	68
6.4.4	Impact of stimulation on test results	69
6.4.5	Impact of the Google Services Framework	71
6.4.6	Comparison of API usage in malicious and benign applications	72
6.5	Discussion	73
6.6	Other experiments	75
7	Limitations and future works	77
8	Conclusions	79
A	List of Dalvik VM instructions	81
	Bibliography	91

List of Figures

2.1	Taint propagation in TaintDroid	7
2.2	Droidbox post-processing	8
2.3	DroidScope system architecture	10
2.4	CopperDroid architecture	11
3.1	Android processes startup	16
3.2	Bytecode comparison between Java VM and Dalvik VM	17
3.3	Comparison of Java JAR and Dalvik DEX file	18
3.4	Internal structures in a DEX file	19
3.5	Structure of an ODEX file	21
4.1	System image instrumentation	26
4.2	Application analysis schema	27
4.3	Instrumentation hook	28
4.4	Object pointers with taint labels in TaintDroid	31
5.1	Increased number of registers in a method	34
5.2	Register shifting	35
5.3	Instrumentation points	36
5.4	Internal instrumentation of a method	37
5.5	External instrumentation of a method	38
5.6	Registers usage comparison	39
5.7	Example of instrumented bytecode	40
5.8	Stack trace acquisition	40
5.9	List of white-listed package names	41
5.10	Depth awareness and field filtering in objects serialization	43
5.11	Runtime configuration file example	43
5.12	Example of logs with serialisation	45

5.13	Example of logs without serialisation	45
5.14	Structure of a try-catch in Dalvik bytecode	49
5.15	Class and call graph database schema	52
5.16	Navigable graph of classes and methods	53
5.17	Error message when running <i>dexopt</i>	54
6.1	Interaction between Android components	60
6.2	Excerpt of generated Monkey events	66
6.3	Comparison of DroidBox and GingerBread without stimulation	67
6.4	Comparison of DroidBox and GingerBread with stimulation	67
6.5	Comparison of DroidBox and GingerBread GSF without stimulation	68
6.6	Comparison of DroidBox and GingerBread GSF with stimulation	68
6.7	Comparison of DroidBox and ICS without stimulation	69
6.8	Comparison of DroidBox and ICS with stimulation	69
6.9	Comparison of DroidBox with and without stimulation	70
6.10	Comparison of GingerBread with and without stimulation	70
6.11	Comparison of GingerBread GSF with and without stimulation	71
6.12	Comparison of the systems with no stimulation	71
6.13	Comparison of the systems with input stimulation	72
6.14	Impact of the Google Services Framework with no stimulation	72
6.15	Impact of the Google Services Framework with input stimulation	73
6.16	Comparison of DroidBox and GingerBread on malware samples	73
6.17	Comparison of DroidBox and GingerBread on benign samples	74

List of Tables

5.1	List of optimised instructions	51
5.2	Example of configuration file listing the APIs to instrument	53
6.1	Malware samples	64
A.1	Dalvik instruction set	82

Chapter 1

Introduction

In recent years *smartphones* had an explosive growth, with Android as their leading operating system[Fortune Tech(2011)]. The coupling of this fact and the sensitivity of the data a *smartphone* usually holds attracted interest of malicious software authors[US Computer Emergency Readiness Team(2011)].

Protecting users from malicious applications is not an easy task, because it typically involves different analysis techniques, aimed at detecting malicious behaviours in unknown applications. One of these techniques is *dynamic analysis*, the process of running an application in a specifically crafted environment that traces the execution and detects given behaviours.

We refer to behaviour as sequence of one or more of events that lead to a specific action. A simple example of behaviour is when an SMS gets sent, a more complicated one may be the combination of actions such as reading data from a file and sending it over the network representing a data transfer. *Dynamic analysis* even being a powerful approach suffers a major problem: limited code coverage, this is due to the fact that for an application to trigger a behaviour it may require specific input and environment characteristics that are difficult to generate automatically[Egele et al.(2008)]. While there exist previous works trying to address the problem of the input to feed to an application in order to trigger a behaviour, it is not common for ANDROID *dynamic analysis* environments to be customisable with respect to system configuration. A customisable environment makes it more difficult for malware to detect whether it is being run in an analysis environment or not. Such a detection is typically part of malware evasion techniques that may be adopted by malicious applications to hide malicious behaviours when they are executed

in analysis environments.

Dynamic analysis environments, also known as *sandboxes*, that are present in state of the art of Android application analysis, still do not focus on an easy configuration of the environment. As a matter of fact, while some of the existing approaches would allow a customisation of the environment, their implementation usually lacks in it, being limited in terms of Android version support, platform architecture target and they typically run only in emulated environments with just a few exceptions.

This work describes DROIDSAGE, a system able to generate *dynamic analysis* environments, starting from the *system image* of a generic Android device or emulator, which is a snapshot of the device hard disk containing the core files used by the ANDROID operating system.

Android applications are executed by means of the DALVIK VIRTUAL MACHINE[Bornstein(2008)], which provides functionality comparable to the JAVA VIRTUAL MACHINE but it has been designed specifically for low powered devices. Similarly to the JAVA VM, the DALVIK VM interprets instructions in the form of bytecode and it also able to interact with libraries containing native assembly code specific to the device through a JNI¹ interface.

In order to give the *system image* the ability to trace specific APIs invocation so that it would be possible to detect given behaviours, we analyse the *Dalvik bytecode*² of all the included applications and libraries. Then we modify the bytecode injecting instrumentation code, that means inserting an invocation before or after an API is triggered that may notify the fact that a behaviour has been triggered.

Since Android 2.3 Gingerbread in 2010, Android system images started shipping their core applications in a format known as ODEX (Optimised DEX). This optimised bytecode version brings a new challenge in the instrumentation of Android applications and its framework. The instrumentation of ODEX files allows to deal with real world Android versions such as physical devices for which is not available the whole source code, but only the base system image.

The instrumentation is performed in such a way to avoid breaking the *signature verification* mechanism of both the applications and the base system. To this end we take advantage of an Android design pitfall that involves op-

¹<http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html>

²<http://milk.com/kodebase/dalvik-docs-mirror/docs/dalvik-bytecode.html>

timised bytecode: we generate instrumented ODEX files that are assumed by the Android operating system to be trusted and are not subject to a proper signature validation.

Differently from other existing approaches, DROIDSAGE also has the unique feature of being able not only to monitor API invocations in third party applications installed, but also to trace applications and libraries shipped with the *system image* itself in the case only the ODEX is available.

In summary, the contributions of our work are:

- Generation of ANDROID *sandboxes* by means of instrumentation Dalvik bytecode in generic *system images*.
- Compatibility with potentially every Android version and device running it, since we modify already working environments. Resilience to malware evasion techniques also benefits from this fact since a different environment is generated for each system image, thus with different characteristics.
- Ability to instrument optimised Dalvik bytecode present in ODEX files.
- Ease of customisation, it is in fact just matter of changing or adding a single line in a configuration file to instrument and monitor a new API.

Chapter 2

Background and motivation

2.1 Android malware

The amount of ANDROID operating system devices and apps has been skyrocketing since 2008 when it was first released and in 2011 the number of *smartphones* sold surpassed the one of personal computers[Fortune Tech(2011)]. The computing power of mobile devices is also getting every month more powerful as proven by devices such as quad-core *smartphones*. In this scenario, the openness of ANDROID system with regard to ease of apps distribution compared to other mobile operating systems has been a playground for malicious software authors aiming to harvest data and steal information from unaware users. The ANDROID permission model, which asks for coarse-grained permissions only at application install time, might also be considered one of the reasons behind the ANDROID malware problem[Felt et al.(2011)]. In fact it is easy to fool a user into accepting broad permissions for an unknown application while it is being installed.

Existing approaches employed by security experts to analyse Android applications and detect malicious behaviours have to cope with huge numbers of applications. It is in fact practically impossible to inspect each application manually, raising the need of mechanisms to automate the analysis process. White-box approaches, those that require prior knowledge for the analysed application, are not suitable in this scenario, that instead requires black-box techniques to analyse a generic unknown application. Black-box analysis of an application can be further divided in two main categories, static analysis and dynamic analysis. The first analyses the application code searching for

specific patterns as an evidence of possible malicious behaviours without actually running it. The second instead executes the application and detects at runtime malicious behaviours when they are triggered.

Static analysis has two main problems: susceptibility to code obfuscation techniques and lack of information only available at runtime that therefore can not be analysed (i.e. remotely downloaded code)[Egele et al.(2008)]. Dynamic analysis, on the other hand has a global view of the system while analysing the application, but it may suffer in terms of code coverage. This is particularly true when speaking of mobile applications that usually require high amounts of user input in order to trigger specific behaviours.

Dynamic analysis, executing the application, has the great advantage of actually proving that an event has happened, while with static analysis it is only possible to pinpoint execution flows that may lead to an event.

Since dynamic analysis requires a specifically crafted environment to perform the analysis, it has to be taken into account that the executed application may try to detect whether it is being run in such an environment by looking for specific characteristics. This process typically known as “fingerprinting” ranges from checking environment variables, to enumeration of device hardware, to network properties and finally to detection of virtual machines.

2.2 State of the art

In this section current existing works in black-box application analysis for the Android operating system are presented.

2.2.1 TaintDroid

The first most relevant work in Android black-box application security analysis can be considered TAINTDROID[Enck et al.(2010)], a dynamic analysis platform focusing on taint tracking of sensitive information. Taint tracking is a methodology typically used to follow information flow as it propagates during execution, this is done by “tainting” object references with specific marks.

The implementation of TAINTDROID consists of a set of patches to a very limited set of specific Android versions (currently Android 2.3 and 4.1), these patches add taint tracking functionality to follow information flow of SMSes, contacts, file contents and other PII (personal identifiable information). The

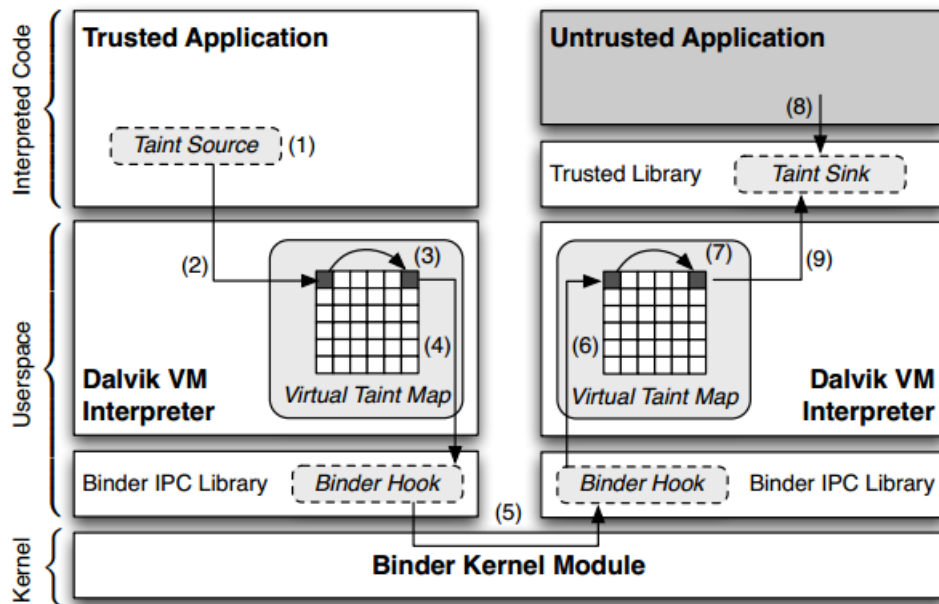


Image taken from [Enck et al.(2010)].

Figure 2.1: Taint propagation in TaintDroid

way TAINTDROID is actually implemented built makes it difficult to modify or port it to other Android versions.

Another issue of TAINTDROID is the fact that taint tracking adds a significant overhead, increasing the time required to perform an analysis. The authors tried to address this problem using taint tracking only in specific code locations, which resulted in lower detection rate.

Nonetheless TAINTDROID suffers the typical issue of taint tracking, that is indirect information flow[Cavallaro et al.(2008)].

2.2.2 DroidBox

DROIDBOX¹ is a variant of TAINTDROID that adds logging not only in case of taint tracking, but also when specific APIs (Application Program Interfaces) are invoked. Post-processing tools are also part of the project, such as a graphical representation of performed activities.

The key feature of DROIDBOX is ease of use, it is in fact ready to be used once downloaded, it comes with a prebuilt system image for the Android

¹<https://code.google.com/p/droidbox/>

```

root@thinkpad: /home/mohsin/DroidBox
^C [*] Collected 22 sandbox logs

[Info]
-----
File name:      ./Samples/HippoSMS/2.apk
MD5:           1a4fb41b95c3cbab05630ee966043c9e
SHA1:          bd7e85f5a0c39a9aecc05dbc99a9e5c52150ba6
SHA256:        1ccb715f38b0c651c927b43c5a28bc6042c0c54654d4b5d071cab94e7b34166e
Duration:      31.1066329479s

[File activities]
-----

[Read operations]
-----
[0.0755288600922]      Path: /data/data/com.android.calendar/shared_prefs/_has_set_def
ault_values.xml(
Data: <?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
<boolean name="_has_set_default_values" value="true" />
</map>

[0.0851199626923]      Path: /data/data/com.android.alarmclock/shared_prefs/AlarmClock
.xml,
Data: <?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map />

```

Figure 2.2: Droidbox post-processing

emulator and scripts to automatise the emulator start, application installation and input generation to stimulate the application under test.

Being based on TAINTDROID, DROIDBOX suffers the same limitations: it is limited to the only Android 2.3 and it is difficult to be extended and modified.

Between its added features, DROIDBOX offers a prebuilt system image, that is an image of the system built from the Android original source code with the TAINTDROID patches applied and the added logging. The image is then obtained as a result of the source code compilation, and that is offered to end users ready to be downloaded and run with the Android emulator.

The usage of this prebuilt image is pretty straightforward for a user, but on the other hand it has drawbacks such as being easy to fingerprint. In fact the same image could be downloaded and analysed by a malware author and specific details could be used to detect if an application runs in DROIDBOX.

With regard to the input generation, DROIDBOX uses MONKEY, a tool part of the Android SDK (Software Development Kit) that generates random

input events.

2.2.3 Andrubis

ANDRUBIS² is the Android subproject of ANUBIS[Bayer et al.(2009)], the online available sandbox realised by Technical University of Wien. Being an online sandbox means that a user just needs to browse the website, upload an application to analyse and wait for a report to be generated. ANDRUBIS takes advantage of both static and dynamic analysis. The static analysis part is performed by means of ANDROGUARD³, an open source tool that provides automated static analysis of Android applications, while the dynamic analysis is backed by a modified version of DROIDBOX.

Since it is based on DROIDBOX, ANDRUBIS performs analysis only on an emulated Android 2.3 platform.

Being accessible only through the website, ANDRUBIS is very useful for end users interested in having a general analysis of an application, but it is not so useful when it comes to security experts interested in specific behaviours. In fact ANDRUBIS does not permit to customise the analysis process.

2.2.4 DroidScope

DROIDSCOPE [YAN AND YIN(2012)] is the first Android sandbox that performs virtual machine introspection. This means that while the application is being run in an emulated environment, the analysis is performed within the emulator itself. The application being run has no access to emulator internals and thus it is not able to tell whether the system is under analysis or not.

As said, the analysis is performed outside the execution environment, this allows the analysis process to be completely external with respect to the execution environment^{2.3}.

DROIDSCOPE is meant to be extended with plugins, its base system in fact only allows obtaining the list of each executed Dalvik VM instruction.

While this choice makes it potentially a good platform to perform black-box application analysis, it is also true that the complete absence of available plugins makes it really difficult to use in practice.

²<http://blog.iseclab.org/2012/06/04/andrubis-a-tool-for-analyzing-unknown-android-applications-2/>

³<https://code.google.com/p/androguard/>

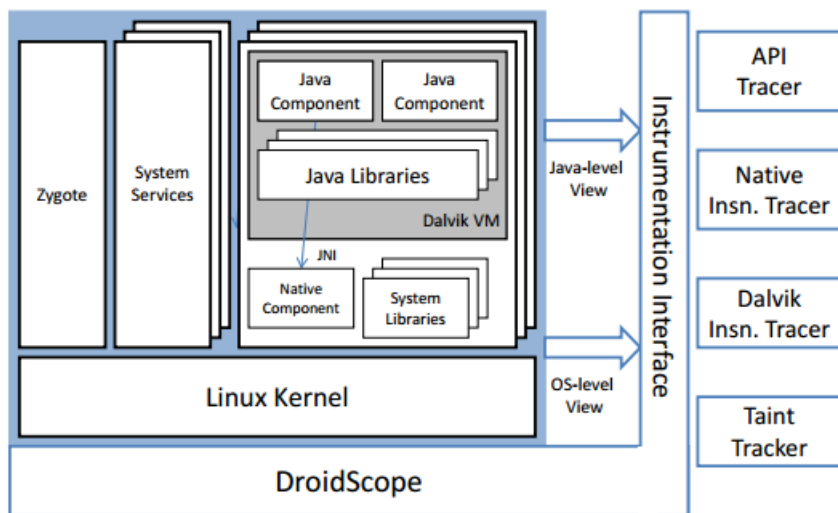


Image taken from [Yan and Yin(2012)].

Figure 2.3: DroidScope system architecture

Another important limitation of DROIDSCOPE is that it is currently compatible only with Android 2.3 and that it requires a custom system image to be built from source. It is not possible to use an existing system image as execution environment instead.

2.2.5 CopperDroid

COPPERDROID [Reina et al.(2013)] has an approach similar to DROIDSCOPE in the sense that it also employs virtual machine introspection, but instead of monitoring the whole system it focuses on the Android BINDER component (Figure 2.4), that is responsible for dispatching Intents. An Intent is a serialisable object representing a message that is used for IPC (interprocess communication) in Android.

This sandbox also tries to increase code coverage searching for Intent receivers, that are the components receiving IPC messages, and generating Intents trying to exercise the receiver functionalities.

COPPERDROID is a lightweight and effective black-box analysis platform, but it is not open source and it is only known to run on Android 2.3.

Similarly to ANDRUBIS, COPPERDROID is available as a web service and thus freely accessible for non technical users.

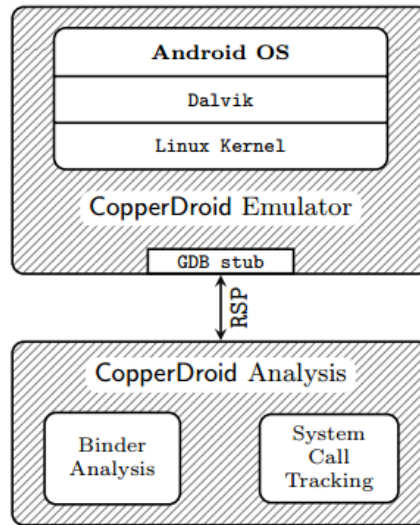


Image taken from [Reina et al.(2013)].

Figure 2.4: CopperDroid architecture

2.2.6 API Monitor

API MONITOR⁴ has a completely different approach compared to the previously described application analysis platforms. It does not require a custom built system image, it focuses instead on the application itself. API MONITOR performs static analysis on an application looking for specific APIs invocations, then it modifies the bytecode of the application injecting instructions to log activity.

API MONITOR has a very flexible approach, but it also has very strong limitations. First and foremost it modifies the application which means that signature verification will fail for the new modified code, this is partially solved by re-signing the application but if an application performs a simple check it will detect the modification.

A second issue of API MONITOR is that it relies on static analysis to modify the application, which means that it is susceptible to code obfuscation techniques that hide API invocations as well as to dynamically loaded code (i.e. downloaded at runtime).

⁴<https://code.google.com/p/droidbox/wiki/APIMonitor>

2.2.7 Apps Playground

PLAYGROUND [RASTOGI ET AL.(2013)] is a full featured black-box analysis platform for Android, it performs taint tracking using TAINTDROID, it monitors sensitive APIs similarly to DROIDBOX, it also performs signature checking against known kernel exploits. PLAYGROUND differently from all the other approaches hereby described includes an advanced input generation system that is window-aware, it in fact requires the Android debug service VIEWSERVER to be enabled and this is used to detect graphic components such as buttons and text fields. The system also performs window-equivalence checks to infer a graph of windows that represent the path between a window and the next.

However, while this work looks promising, the system has not been made available, and it is thus not usable by researchers.

2.3 Goals and challenges

Given the limitations highlighted in Section 2.2, our work aims to improve current dynamic analysis state of the art for the Android operating system proposing an approach that allows to generate customisable Android sandboxes targeting multiple versions of the operating system and not limited to emulated environments.

What we are interested in building is an Android black-box dynamic analysis sandbox. The target of this system is a generic unknown application, for which no prior knowledge is available. This requirement excludes white-box analysis approaches from the scope of this research. The system can be configured to trace specific behaviours, allowing a user to trace only the behaviour he or she is interested into. These behaviours are detected by means of dynamic analysis techniques when they are triggered at runtime.

DROIDSAGE aims to propose a very flexible approach, generating dynamic analysis environments in which a configurable set of API are monitored and instrumented. Specifically targeting only Dalvik bytecode it is possible for our approach to keep compatibility with potentially every Android version. For the same reason, it is also possible for DROIDSAGE to target different platform architecture.

Differently from other approaches, which typically force to build a new system from source, this work permits to modify existing system images. Dynamic analysis techniques are thus introduced in pre-existing systems through

instrumentation of the Dalvik bytecode. Real devices can also be targeted starting from their system image.

Currently, most of the Android devices run on *ARM* processors, but *x86* and *MIPS* Android variants are also available. Our work aims at being compatible with each of these versions, this is possible since instrumentation only happens at bytecode level.

With this high compatibility, it is possible to use our system to test the same application on multiple environments, allowing to spot differences between the different executions that may be due to the underlying system. This may be useful when detecting which are the target platforms for specific applications, such as defining the target user base of a specific malware application. Furthermore, allowing such a flexible environment implies a decreased detectability of the analysis platform, thus being more resilient to malware evasion techniques, that try to masquerade malicious behaviours from being triggered in analysis environment.

Instrumented bytecode is optimised before being deployed in the end environment. This step allows avoiding to break signature verification for applications and libraries present. As described in Section 3.5 *signature verification* only happens on an application or library original content, and not on the optimised version of its bytecode, which only contains a hash sum which refers to its unoptimised original form. Even parsing the instrumented bytecode, it would be difficult to detect the presence of the injected bytecode.

Chapter 3

The Android runtime

In this section a brief description of the Android runtime components is given, with information relative to the Dalvik virtual machine and its bytecode, as well as a quick explanation of the signature verification mechanism.

3.1 The Dalvik Virtual Machine

The Dalvik virtual machine allows running applications in the Android operating system. Similarly to the most known Java virtual machine, it offers a set of predefined instructions that allows applications to perform more complex tasks and computation.

The Dalvik instructions are designed to be used in an object oriented scenario, exposing instructions that interacts with objects fields and methods.

The sequence of instructions that compose a method are described by a series of bytes representing the equivalent Dalvik bytecode for each instruction. In the bytecode format, references to object classes, methods and fields occur by index for performance and space reasons. These indexes represent the entry number in the respective method table, field table or class table.

Those tables, together with the encoded instructions and additional data are stored in a Dalvik EXecutable file (DEX) as shown in Section 3.3.

DEX files represent the core of each Android application, in fact the classes.dex file contained in an Android application is loaded by the Dalvik virtual machine when starting the application.

The Dalvik VM has a peculiar way to start a new application, in fact only one process is started at boot time loading the VM, then for each application to

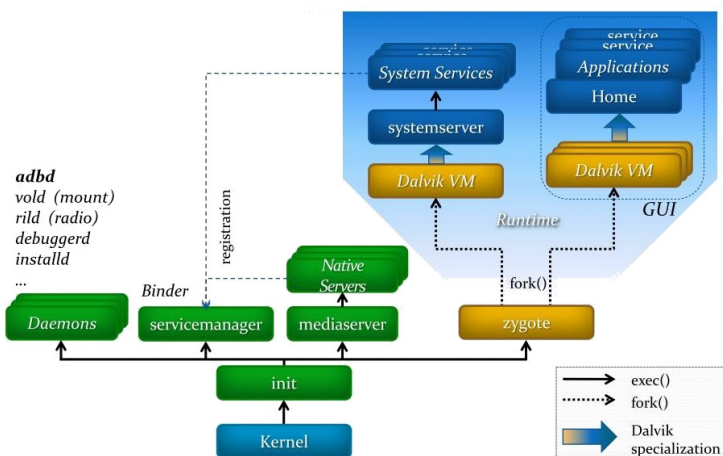


Figure 3.1: Android processes startup

run, that process forks, switches its user and group id and loads the additional application bytecode to be run (Figure 3.1).

Other than Dalvik bytecode, the Dalvik VM also interacts with native libraries through Java Native Interface (JNI). In this case such libraries are Executable and Linkable Format (ELF) files, containing native assembly for the specific architecture underlying the Android operating system.

Those libraries expose functions that will be invoked by the Dalvik VM as if they were normal class methods, passing Java objects references as parameters.

Native libraries code has then the ability to invoke regular bytecode methods using the JNI architecture.

3.2 Comparison with the Java Virtual Machine

The Dalvik virtual machine can be considered similar to the Java virtual machine in the sense that they both offer a way to execute bytecode that allows invoking methods on objects, access their fields and offering computation such as arithmetic instructions.

The underlying structure of the interpreter is instead very different. The most important difference is the fact that the Java VM is stack based, while the Dalvik one is register based (Figure 3.2).

The Java VM extensively uses instructions such as pop and push to place and get values from the stack, the instructions are in fact required to mainly use values placed on the stack and return an operation result on the stack too.

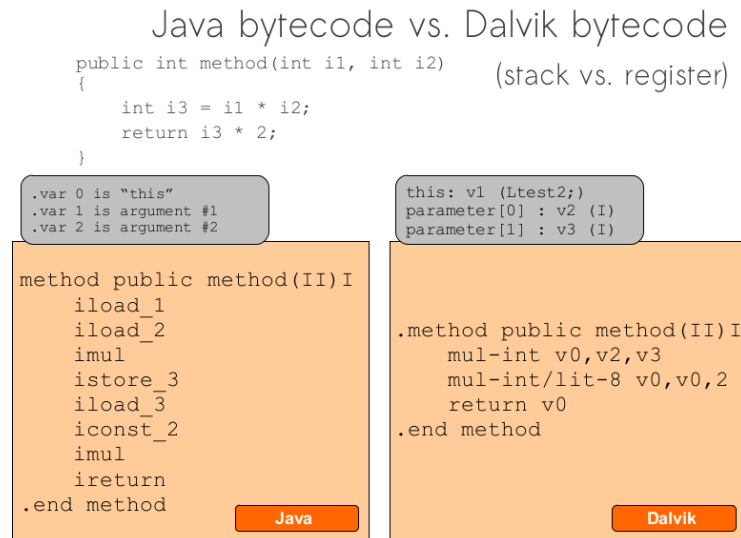


Figure 3.2: Bytecode comparison between Java VM and Dalvik VM

A simplification allows the definition of local variables, which still have to be pushed on the stack each time before being used.

What happens with the Dalvik VM is instead very different, it has no stack at all. This VM statically defines how many registers each method will use, those registers can be considered as local variables that can be directly accessed both for reading and writing values.

Both the approaches have pros and cons, what is generally described comparing the two virtual machine is that the Java VM requires a high number of single instructions to achieve a result, this is due to continuously having to read and write values to and from the stack. The Dalvik VM instead does not have to perform pop and push operations since there is no stack at all.

On the other hand, the Dalvik VM pays the price in terms of code size, in fact each instruction in the Dalvik VM is typically wider since it has to specify which registers to use.

Another interesting difference is the fact that the Java VM keeps the bytecode of each class in a separate file, grouping them together multiple classes in a single JAR file. The Dalvik VM instead groups together classes in a single bytecode file, minimising the space required by the constant pools, which is then shared by the classes (Figure 3.3).

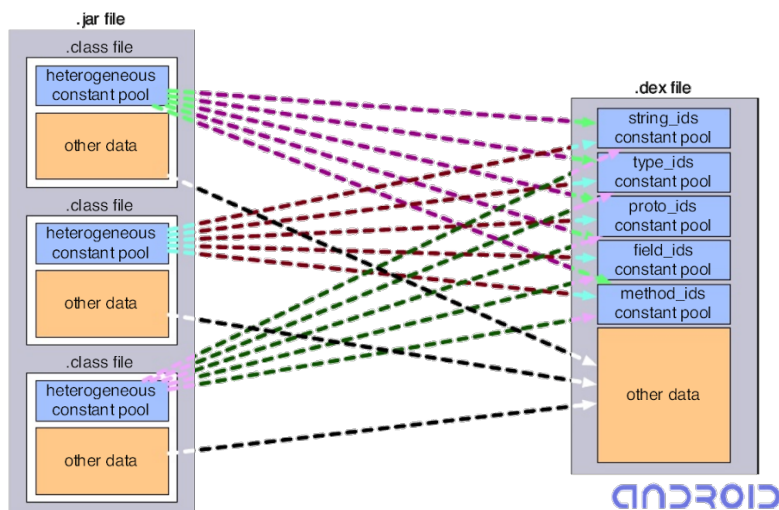


Figure 3.3: Comparison of Java JAR and Dalvik DEX file

3.3 Structure of a DEX file

A DEX file contains bytecode loaded by the Dalvik virtual machine. In this format a single file is composed of a set of sections, each one tightly connected to the others.

The core element in this file is the bytecode representing each of the instructions in the sequence specified by the method code that will be executed by the Dalvik VM.

In order to represent the bytecode for a specific instruction, other references have first to be defined. First of all, each instruction is identified by an opcode, then depending on the parameters the instruction has to specify, different instruction formats have been defined in the Dalvik VM.

Typically an instruction can refer to a class, a method or a field in a class, or a value which may be embedded in the bytecode itself or obtained from a register at runtime.

Classes, methods and fields are referred by index in the respective table, each one contained in a different section of the DEX file (Figure 3.4).

There are two different classes tables, one listing all the classes referred (used), while another table only contains the classes defined in the current DEX file. This second table for each entry has the list of the methods and fields the class defines.

With respect to objects and fields values, these may be represented in the

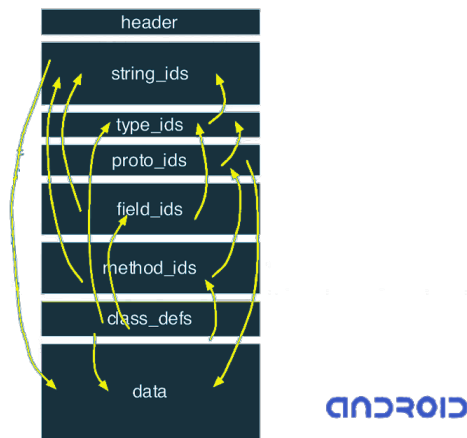


Figure 3.4: Internal structures in a DEX file

bytecode itself. For instance when initialising a boolean variable or an integer one, their values can be embedded in the bytecode as a raw value expressed in the instructions parameters. Something different happens when dealing with string values, DEX files in fact have a specific section dedicated to string values, this is referred not only by values to assign to string objects in the bytecode, but it is also used to declare values such as class names, method names or field names that will be referred in the others sections of the DEX file.

A highly indexed structure as the one here described strongly reduces the size of the constant pool the instructions have to refer to. This is an important feature for low powered devices, where the memory footprint has to be low (Figure 3.3).

3.4 Structure of an ODEX file

An ODEX file is obtained through the process of optimisation of a DEX file. The optimisation is performed by the dexopt binary that comes with the Android operating system.

The most significant changes that this phase brings to the bytecode are that methods and fields will be accessed by index in the method and field tables which are already available at runtime, avoiding the lookup by method or field index in the respective section of the DEX file.

To ensure that indexes are correct, a hierarchy of ODEX files is used,

specifically the Android operating systems uses and environment variable, defined as `BOOTCLASSPATH`. This variable contains a list of paths separated by a colon mark, each one referring to an ODEX file.

There is a strict ordering in the entries listed, so that an ODEX file declares which are its dependencies, declaring their absolute path and their *sha-1* signature. This way allows detecting whether one of the referenced dependency has been modified and avoid loading tampered code.

The structure of an ODEX file (Figure 3.5) is composed of a header, a section containing an embedded DEX file, and three other sections: one representing a class hash table to speed up access to class definitions, another section holding the references to the dependencies, and a last section containing register maps that are used to ease the garbage collection at runtime.

The embedded DEX file is basically the original DEX file with a few changes: instructions used to access object methods or fields are optimised wherever possible. Some other differences may apply in the case the underlying architecture does not use little-endian as byte encoding.

The optimised index of a method refers to the index of the method in the virtual table of the object that is referred as runtime.

This virtual table is built starting from the one inherited by the parent class, then all virtual methods are added, in the order they appear in the method table for that class in the DEX file.

Virtual methods are those methods that are not static, private or constructors ¹.

Regarding to field index, a field table for the object is kept at runtime by the Dalvik VM, this first includes the field table of the parent class, then all the instance fields of the local class are added. None of the static fields reaches the field table.

The fields added to the table are subject to a particular sorting algorithm, the idea is to first list object types (non native types) fields, then wide fields (double or long values) and then all the remaining instance fields.

The way the field list is rearranged starting from the original one in the DEX file, tries to minimise the number of shifts or swaps needed to build the final list.

¹<http://milk.com/kodebase/dalvik-docs-mirror/docs/dex-format.html>

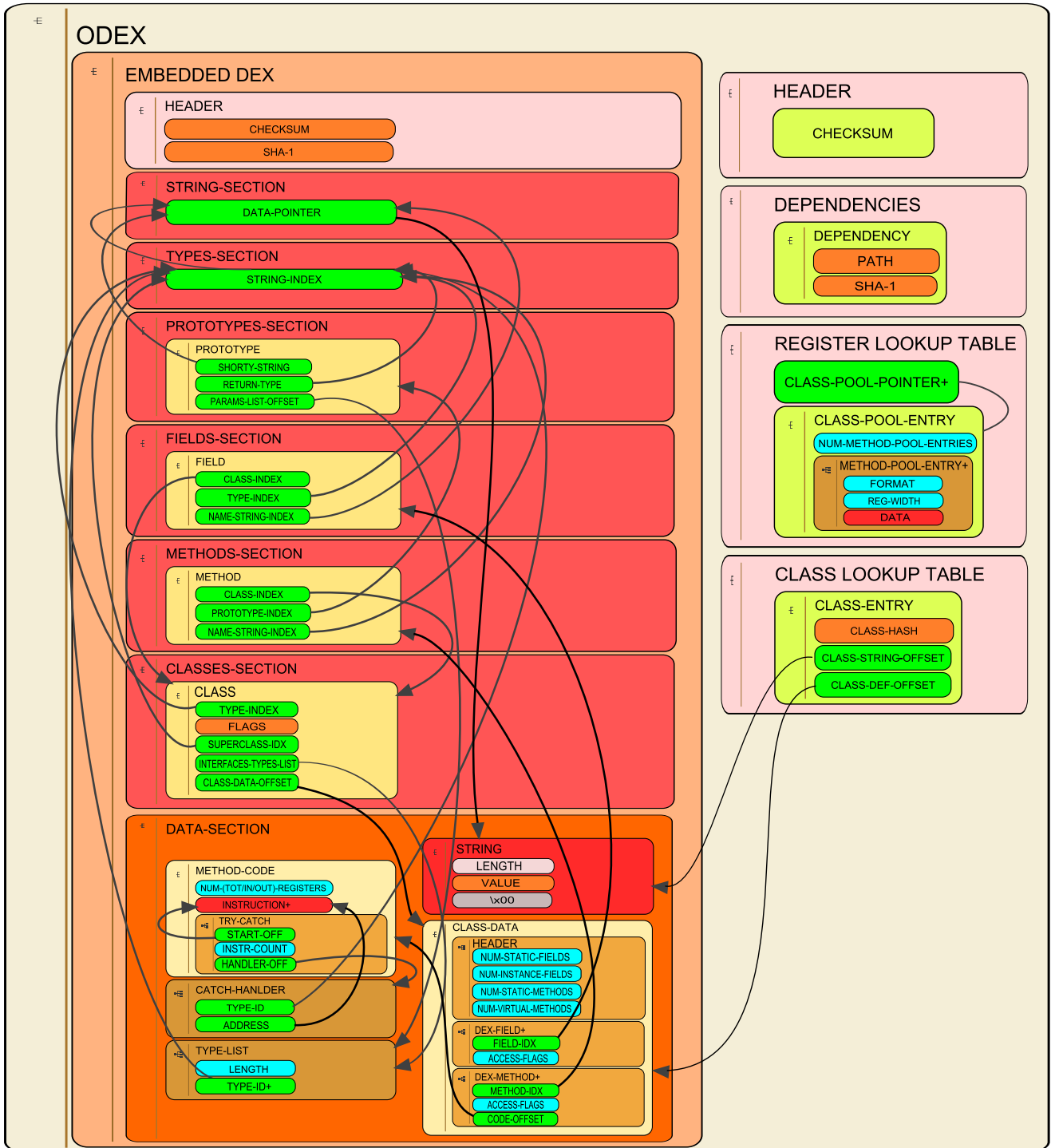


Figure 3.5: Structure of an ODEX file

A first step is done in order to first list all the object types fields: a first index is used starting from the first entry in the list. If it is an object type reference, it is left there, otherwise a second index is used, starting from the last entry and decreasing the index, if it is an object reference then a swap occurs, the first index increased and the second one decreased. Otherwise just the second one is decreased.

When the two indexes reach the same value, the iteration is stopped.

Wide fields, such as double or long values have to be 64 bit aligned, thus depending on the underlying architecture, a non wide field might be used as filler to perform the alignment.

A similar process to the one described for sorting object references first is then applied on the remaining sublist for wide fields.

The hereby described method virtual table and field table is then referred by the optimised instructions, such as using `invoke-virtual-quick` instead of the regular `invoke-virtual`.

For what concerns the other sections in the ODEX file, a list of the ODEX dependencies is present in a separate section, including absolute path and *sha-1* hash of the referred file.

Two other sections are present one containing a hash table to speed up the access to a class definition in the embedded DEX file, and another one used by the garbage collector to optimise the memory management.

3.5 Signature verification in Android

Android applications are shipped as ZIP files containing at least an `AndroidManifest.xml` file and a `classes.dex`. To be able to install an application on an Android system it is also required that the application has been signed with a valid certificate.

The signing process is based on the S/MIME standard [Ramsdell(2010)], that permits authentication and integrity to be verified.

The signed ZIP file contains a `META-INF` folder, with the following files inside:

`MANIFEST.MF` Contains the list of the other files in the ZIP archive except the ones in the `META-INF` folder. For each of the files its hash is computed, encoded using base64 algorithm and added to the list together with the relative path in the archive.

Algorithm 3.1 Manual verification of an APK S/MIME signature

```

$ openssl asn1parse -inform DER -in cert.rsa -i -dump
  0:d=0  hl=4 l= 772 cons: SEQUENCE
  4:d=1  hl=2 l=   9 prim: OBJECT                :pkcs7-signedData
  ...
 77:d=7  hl=2 l=   9 prim: OBJECT                :sha1WithRSAEncryption
  ...
$ openssl smime -verify -noverify -inform DER -content cert.sf -in cert.rsa
-out /dev/null
Verification successful

```

CERT.SF This file is similar to **MANIFEST.MF**, it also contains the list of the files with a hash for each of them, but the hash is computed differently: instead of being computed on the file itself, it is computed by hashing the strings of the respective file lines in the **MANIFEST.SF**.

CERT.RSA This is the actual digital signature of the package. It is in fact an S/MIME signature of the **CERT.SF** file, with information on the certificate used to sign it, that typically is self signed.

The certificate used to sign the application is then used to allow inter-application permissions, for example allowing interaction only between the applications signed with the same certificate. This last feature may be enabled in the **AndroidManifest.xml** file with the *protectionLevel* attribute ².

A manual way to verify an application signature is to first verify the **CERT.SF** against the **CERT.RSA** digital signature (Algorithm 3.1). If they match, proceed verifying the hashes between **MANIFEST.MF** and **CERT.SF**. If they match as well, the sha1 hash sum of the real files can be computed and verified.

Once verified the digital signature of the application, the installation phase may proceed. This includes the copy of the APK package to */data/app* when installing in the internal storage or */mnt/asec* when installing on external drives such as SD cards. Extraction of resources files from the application package is then performed to the */data/data* folder.

At this point optimisation of the Dalvik bytecode included in the application is performed by means of *dexopt*. The resulting optimised bytecode is

²<https://developer.android.com/guide/topics/manifest/permission-element.html#plevel>

then stored in the `/data/dalvik-cache` folder until the application is uninstalled or an update of the system may execute the wiping of the cache.

Verification of the optimised bytecode is performed at load time by the Dalvik virtual machine before the application is actually run. This validation phase does not include digital signature verification, but only a checksum comparison.

The ODEX file with the optimised bytecode contains a list of dependencies with the respective path and *sha-1* hash to verify the dependency files have not been modified and an *Adler32* checksum of the ODEX sections after the embedded DEX is placed in the ODEX header. The embedded DEX on its own has both an *Adler32* checksum and *sha-1* hash in the DEX header.

It is worth to notice that bytecode in an ODEX file is not properly verified against the original DEX bytecode to ensure the code logic has not been changed. This is even impossible to be performed in the case the ODEX file fully substitutes the unoptimised version, that is what normally happens in Android system images. Such a phenomenon is normally justified by disk space constraints, in order to avoid wasting space carrying the original DEX file if it will not be used to actually run the applications.

The approach described in this work takes advantage of this glitch to bypass signature verification of the instrumented bytecode.

Chapter 4

Approach

In this chapter insights of the realised system are described in Section 4.1, along with a description of the usage of the instrumented system image to analyse an application (Section 4.2) and an example of actions to be performed when the instrumented APIs are invoked at runtime (Section 4.3). Finally a comparison with some of the other currently existing solutions is presented.

4.1 System image generation

Our approach allows the generation of Android system images to be used as black-box dynamic analysis platforms, this is done executing a series of tasks (Figure 4.1) that can be summarised as follows:

- 1) **File extraction** The provided system image is read and its content is extracted in order to be processed (Subsection 5.4.1).
- 2) **Bytecode identification** Application and libraries containing Dalvik bytecode are identified inspecting the content of the system image (Subsection 5.4.2).
- 3) **ODEX conversion** Once the optimised Dalvik executables are localised, they are processed and optimised instructions are replaced with their non optimised equivalent (Subsection 5.4.3) with the aim of identifying the invoked methods.
- 4) **Bytecode analysis** A call graph has to be generated in order to locate instrumentation points. This is done parsing the bytecode and keeping track of invocations (Subsection 5.4.4).

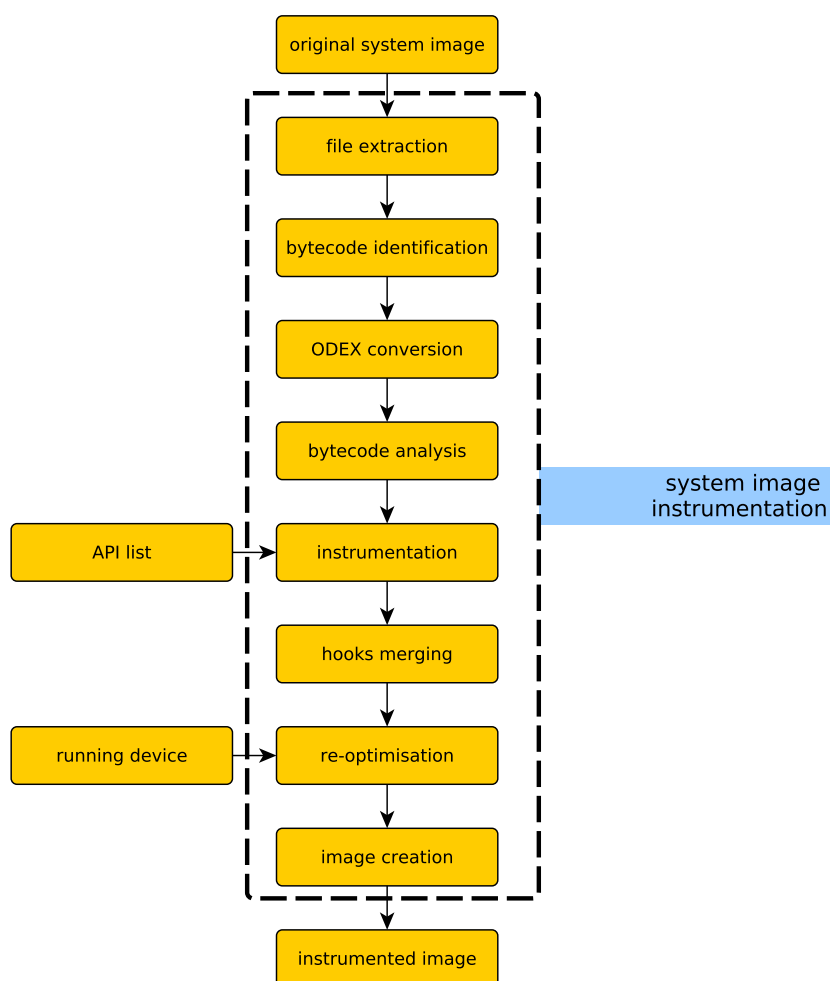


Figure 4.1: System image instrumentation

- 5) **Instrumentation** The Dalvik bytecode is instrumented. The set of instrumented APIs can be fully customised (Subsection 5.4.5), then for each API to instrument there are two possible situations: if the API is directly instrumentable then it is instrumented in its own code, otherwise if it is not directly instrumentable (i.e. interfaces, abstract methods, native methods) invocations are instrumented before or after they are executed (Section 5.2).
- 6) **Hooks merge** Instrumentation hooks (Section 4.3) are merged in the core library to ensure their reachability from other libraries or applications.

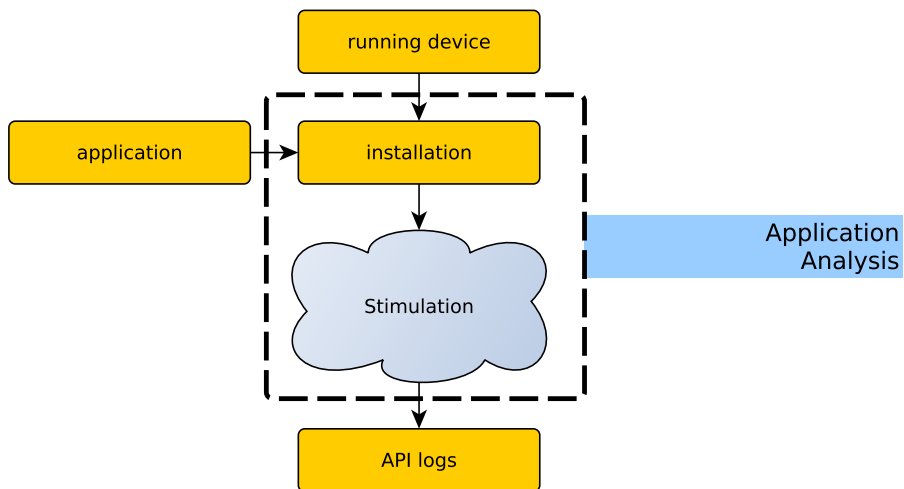


Figure 4.2: Application analysis schema

- 7) **Re-optimisation** The bytecode is re-optimised to avoid breaking the signature verification (Subsection 5.4.7).
- 8) **Image creation** The resulting system image is built with the instrumented bytecode (Subsection 5.4.8).

Each of the listed phases will be described more precisely in the following sections, while their implementation details will be presented in Chapter 5.

4.2 Application analysis

In order to analyse an application (Figure 4.2), this must be installed in the instrumented system image (Section 4.1). This can be performed manually or through ADB. The second option can be used for automated analysis platforms since no manual interaction is needed.

It is then necessary to generate input in order to stimulate the application to trigger each of its behaviours. Although it is not a key feature in our work, an input generation system had to be developed in order to test the system (Subsection 6.1.2).

While being executed in the instrumented environment, the application might reach an instrumented API, and in the case the invocation is found not to be generated from the system itself, then it is logged.

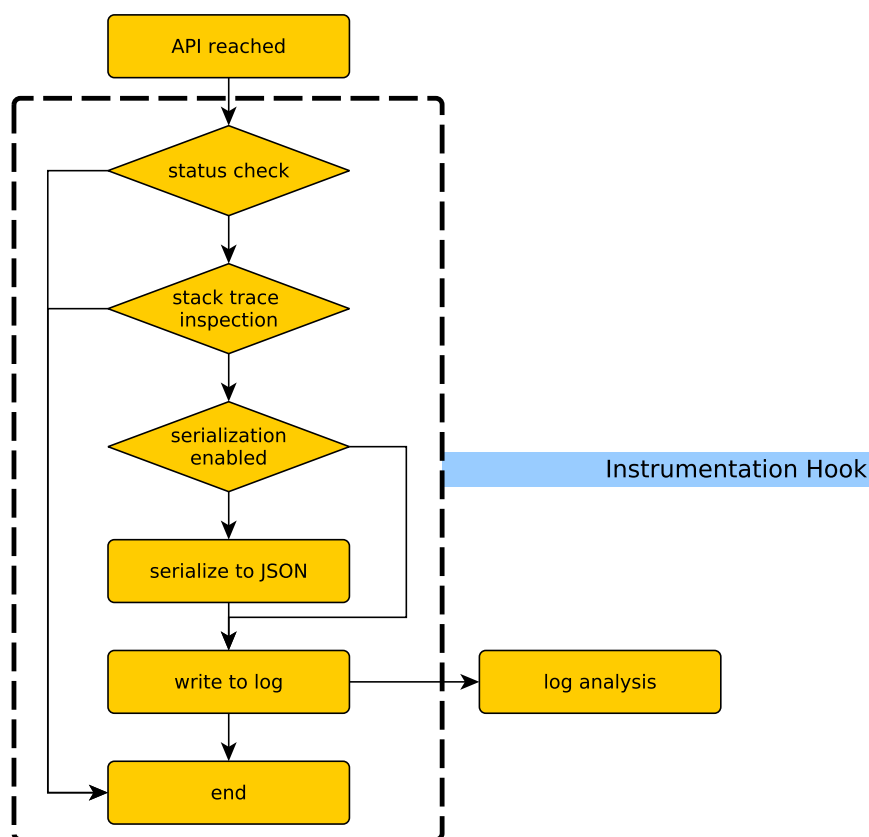


Figure 4.3: Instrumentation hook

This notification process happens through logging functionality built-in in the Android system, the reason behind this choice is ease of use and the fact that it does not require special permissions to be used. The same logging APIs are used in similar works such as TAINTDROID and DROIDBOX.

The resulting notification sent externally through a log entry can then be used to perform behaviour detection, but this is not in the scope of this work.

4.3 Instrumentation hooks

When an instrumented API is reached, multiple tasks have to be performed. To this end, we added instrumentation hooks, implemented as a set of classes and methods that handle API invocations. This is performed as follows:

Status check The hooks regularly check if status flags have been updated.

Flags are on/off switches that enable or disable specific features at runtime. Currently three flags are configurable: enabled flag, stack trace inspection, serialization to JSON. A package name white-list is also used to tell apart system packages from external applications. Both flags and package names white-list are configurable at runtime.

Stack trace inspection If the corresponding flag is enabled, a stack trace inspection is performed starting from the most recent entry. Each entry can be recognised either as being part of the instrumentation framework itself, part of the system framework, or part of an external application. In the first case the stack trace inspection stops, and the *API* is not logged since originating from code that is part of the instrumentation framework. In the second case, the stack trace inspection continue to the next entry. Alternatively, in the last case, the stack inspection stops and the *API* is logged since originating neither from the instrumentation nor the system frameworks, thus it is assumed to be part of an external application.

Serialization Differently from other approaches, since the instrumentation is performed automatically and the target *APIs* are customisable, it is quite difficult to automatically infer what information has to be logged, this normally depends on the semantic of the single *API*. To address this problem, it has been decided to keep a generic approach and serialize to plain text both the *API* parameters and the object on which the *API* has been invoked (reference to *this* object). Different serialization libraries have been tested and for readability purposes *JSON* has been chosen as the preferred serialization output format. The *API* invocation, with its parameters or return value and the eventual *this* object reference, before being logged has first to be serialised. Depending on the *JSON* serialisation flag, the corresponding objects are serialised either in *JSON* or using the *Object.toString()* method which *Java* offers.

Logging The serialised *API* invocation is then outputted using the *Log* class in a synchronised fashion, to avoid overlapping between log entries. A further consideration is made when logging: the standard *Log APIs* only logs up to 4096 bytes. To overcome this limit, multiple invocations to the *Log* method are performed.

4.4 Comparison with other state of the art solutions

In this section further details about the proposed approach will be given with respect to existing solutions described in Section 2.2.

4.4.1 Comparison with API Monitor

The approach technically most similar is API MONITOR, described in Subsection 2.2.6. Bytecode analysis and modification is in fact the core of both these approaches.

API MONITOR directly targets the content of Android applications, and it does not require any special execution environment.

To analyse an application, API MONITOR first disassembles the Dalvik bytecode contained in the application, then the obtained text representation of the bytecode is instrumented injecting log calls. The result of this step is then re-compiled, re-packaged and re-signed in another application. The instrumented application can then be installed and run on a target system.

The process of re-signing the application invalidates the original digital signature distributed with the application [Barrera et al.(2012)], which in turn may cause the application to not behave correctly.

Differently from API MONITOR our work does not target the application bytecode, that instead is left unmodified in order to guarantee signature verification and avoid side issues deriving from its modification. The real target of bytecode analysis and modification in this work is the set of Android system APIs that are instrumented directly in the system image rather than in the application.

Another benefit of our approach is given by having a system wide view, not only limited on the application code.

While it is common practice to use code obfuscation techniques in applications that therefore may cause issues with static analysis, it not common at all to obfuscate the core libraries that come with a system image: in a set of 20 system images analysed none of them has been found having core libraries obfuscated. This ensures higher reliability of the static analysis performed before the instrumentation.

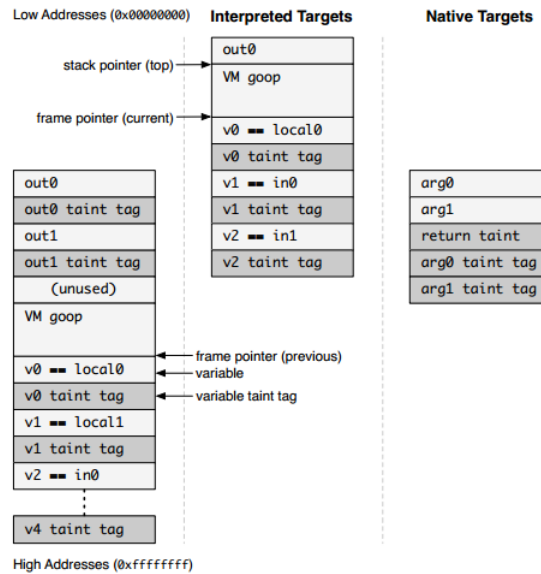


Figure 4.4: Object pointers with taint labels in TaintDroid

4.4.2 Comparison with TaintDroid

TAINTDROID is a milestone in Android security, it is in fact the first known information-flow tracking system aiming to detect privacy leaks in Android applications.

The approach used by TAINTDROID can be considered orthogonal to the one described here. In fact, a TAINTDROID environment's system image can be instrumented with our approach, adding taint tracking functionality in the proposed solution at no cost.

The instrumentation of a TAINTDROID environment has only required to consider altered method and field indexes caused by the modified Dalvik virtual machine that TAINTDROID brings. To keep track of object taints it doubles the object reference size in order to have an extra value for the object taint (Figure 4.4). This is further described in Subsection 5.4.3.

4.4.3 Comparison with DroidBox

Both DROIDBOX and our approach perform instrumentation of APIs. However, while DROIDBOX patches the Android source code before it is compiled, our approach directly modifies the Dalvik bytecode of a generic system image, without requiring a new system image to be built from source.

This is a great advantage when it is not possible to generate custom system images, that is particularly true when speaking of real devices since hardware manufacturers tend not to release open source drivers.

Although the instrumentation performed by DROIDBOX has the advantage of being manually verified, this is also a limitation when the number of APIs to track becomes significant. Manual instrumentation has also an important downside when a patch has to be ported between two different Android version, which may be one of the motivations for DROIDBOX being still limited to Android 2.3.

4.4.4 Comparison with Andrubis

Since ANDRUBIS is based on DROIDBOX, the considerations in Subsection 4.4.3 apply. Specific comparison may be added stating that ANDRUBIS being an online platform which performs its analysis using always the same system image for every test, it may be an easy target for malware evasion techniques. It is in fact quite straightforward to fingerprint the environment and trigger (or not) behaviours on that basis.

The proposed approach makes fingerprinting of the environment a more difficult task, and thus it is more resilient to malware evasion techniques.

Chapter 5

Implementation

In this chapter we present a bottom-up description of the implementation details of our system, ranging from injection of bytecode instructions up to the instrumentation of a whole system image.

5.1 Bytecode injection

This section describes the technique used to inject bytecode in an existing method to perform method instrumentation.

5.1.1 Reasons to inject bytecode

Instrumentation of a method in *Dalvik* bytecode requires the injection of additional instructions in order to invoke the instrumentation hooks with information relative to the traced *API* passed as parameters. Such information has to specify at least an identifier of the target *API* invoked and its parameters or return value. While parameters or return value are already available in registers at instrumentation point in the bytecode, no reference to the method name or class name is present, so this information has to be added.

Since the *Dalvik* virtual machine is register based, we need to use registers to store this data. The number of registers a method will use is declared in the method definition and can not be modified at runtime. To address this, we can either increase the number of register in the method definition, or backup and restore the register values.

#5	: (in Llibcore/io/IOBridge;)	#5	: (in Llibcore/io/IOBridge;)
name	: 'closeSocket'	name	: 'closeSocket'
type	: '(Ljava/io/FileDescriptor;)V'	type	: '(Ljava/io/FileDescriptor;)V'
access	: 0x0009 (PUBLIC STATIC)	access	: 0x0009 (PUBLIC STATIC)
code	-	code	-
registers	: 4	registers	: 6
ins	: 1	ins	: 1
outs	: 2	outs	: 2




Figure 5.1: Increased number of registers in a method

5.1.2 Injection technique

While backing up and restoring registers could be seen as an easy and effective solution, it can not always be applied. Since the *Dalvik* virtual machine allows multi-threading, it is necessary to keep an additional reference in order to restore the correct registers and values. One can think of this scenario as a surveilled wardrobe: when someone deposits a coat a ticket is given him in order to allow him to get back the same coat he left. Similarly if the backup manager has to deal with multiple threads, it is necessary to keep references to pair the saved object to their owners.

In order to use a non-free register it is required to first make a back-up copy of it, but in order to back it up another register holding an identifier has to be used. This causes a loop that does not allow this methodology to be applied when no free registers are available.

A better approach is to increase the number of registers used by the method (Figure 5.1), this can avoid the need to save and restore register values but on the other sides it introduces a new challenge described in the next paragraph. The number of additional registers used in this work has been kept minimal, employing just two additional registers, which is due to the size of wide variables, used to hold *long* and *double* values.

5.1.3 Bytecode adaptation

The registers used by a method are automatically filled with parameters and eventually the reference to *this* object by the *Dalvik* virtual machine when the method is invoked. The registers holding these values are the ones with highest index number, so that in a method with 10 registers and 3 parameters, the registers holding the parameters will be registers r7, r8 and r9 (counting from r0).

Adding new registers in this scenario implies the introduction of regis-

T	r0	r1	r2	r3	r4	r5	r6	r7	r8	r9
0	-	-	-	-	-	-	this	p1	p2	p3
1	-	-	-	-	this	-	-	p1	p2	p3
2	-	-	-	-	this	p1	-	-	p2	p3
3	-	-	-	-	this	p1	p2	-	-	p3
4	-	-	-	-	this	p1	p2	p3	-	-

Figure 5.2: Register shifting

ter renaming or register shifting, since the existent bytecode will try to use different registers from the ones that hold the parameters.

Register renaming is a technique used in computer architecture that basically remaps registers names or indexes without changing the semantics in the flow of operations; this is typically used to increase data locality and minimise the number of assignments to registers [Sima(2000)].

In this case register renaming is not meant to improve the performances, but merely to adapt already optimised bytecode to use different register indexes.

Even renaming all registers indexes in bytecode instructions, would not suffice to make it work properly. This is because most of the instructions available for the *Dalvik* virtual machine only allow a 4-bit addressing, which means that those instructions only work with registers from 0 to 15, this is also described in Section 3.1. When a register index is changed it may be the case that the new index is not addressable with only 4 bits. Although in some cases alternative instructions are available, it is not a viable solution.

The remaining and actually used approach is register shifting, this consists in shifting the registers with the goal of restoring an environment in which the pre-existent bytecode is able to run with no further modification. To this end when an instrumented method starts, the number of registers used is incremented and a prologue of instructions that shifts registers downside for an amount of registers correspondent to the number of added registers is injected. A graphic representation of this prologue is shown in Figure 5.2.

5.2 Bytecode instrumentation

Once the method that is going to be instrumented has been prepared increasing the number of registers and the prologue providing register shifting has

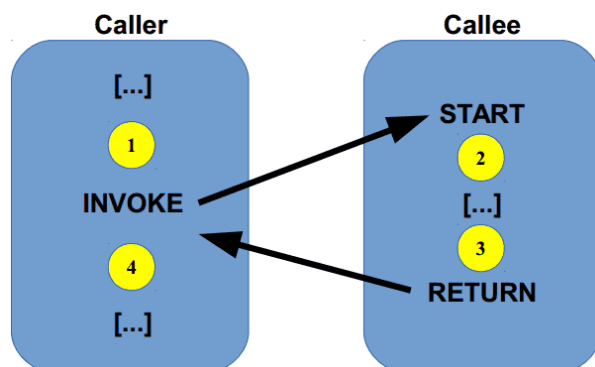


Figure 5.3: Instrumentation points

been injected, the next step is to inject the actual invocation of the instrumentation hook that will notify that an API of interest has been reached.

5.2.1 Instrumentation points

The invocation can occur in four different places as shown in Figure 5.3, which are:

1. Ahead of the method invocation.
2. Inside the method, just when it starts.
3. Before a return inside the method.
4. After the method invocation.

Depending on the characteristics of the *API* to trace, it may not be possible to instrument it directly (points 2 and 3), this happens with methods which do not have bytecode whether because they are abstract or because they are native, thus implemented using *JNI*; in those cases external instrumentation has to be used (points 1 and 4).

Furthermore one could be interested only in the return value of the API to trace or only in the parameters it has been invoked with. This is quite typical when reading from a stream, in such cases, it is in fact only interesting to be notified on the return value, which is what has been read from the stream.

DROIDSAGE instrumentation keeps track of method parameters when instrumenting at point 1 and 2, while tracks the return value when instrumenting in points 3 and 4.

```

public int example(int p1, int p2, String sign){
    if( sign.equals( "+" ) )
        return p1 + p2;
    else
        return p1 - p2;
}

public int example(int p1, int p2, String sign){
    Instrument.call("example", this, p1, p2, sign );
    if( sign.equals( "+" ) ){
        int ret = p1 + p2;
        Instrument.ret("example", this, ret);
        return ret;
    } else {
        int ret = p1 - p2;
        Instrument.ret("example", this, ret);
        return ret;
    }
}

```

Figure 5.4: Internal instrumentation of a method

5.2.2 Internal instrumentation of a method

A method directly instrumentable is a method which comes with its own bytecode, thus not abstract or native. When a directly instrumentable method has to be instrumented, this happens in its own bytecode, specifically this happens at the beginning of the method or before a return instruction (Figure 5.4), depending on the target of instrumentation being the parameters or the return value. In the first case the parameters will be in the highest registers, while in the second case the return value will be the register used as parameter of the return instruction.

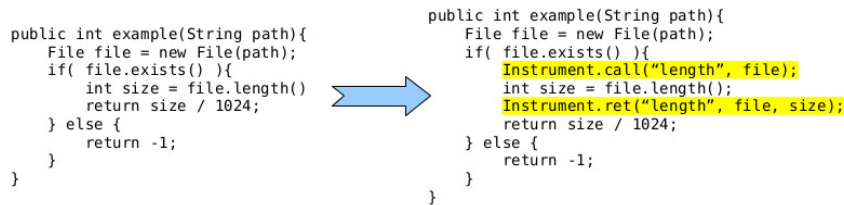
In the case of non-static method, the register holding the reference to the *this* object will be the first register before the parameters. This is always available when instrumenting at method start, but it might be overwritten with other values when the method is about to return. This also applies to method parameters, which might be overwritten as well.

5.2.3 External instrumentation of a method

When a method is not directly instrumentable, its invocations from directly instrumentable methods are instrumented. This means that whenever an invocation of a target API which is not directly instrumentable is found in the bytecode of another method, the invocation itself is instrumented.

The instrumentation can happen before or after the invocation, depending on the target of instrumentation being the parameters or the return value respectively (Figure 5.5). When instrumenting before the invocation, the registers used as parameters for the invocation instructions will hold the method parameters, and in case of a non-static method, the reference to *this* object.

When instrumenting after the invocation, method parameters and the eventual reference to *this* object will still be available, but the return value is



```

public int example(String path){
    File file = new File(path);
    if( file.exists() ){
        int size = file.length()
        return size / 1024;
    } else {
        return -1;
    }
}

public int example(String path){
    File file = new File(path);
    if( file.exists() ){
        Instrument.call("length", file);
        int size = file.length();
        Instrument.ret("length", file, size);
        return size / 1024;
    } else {
        return -1;
    }
}

```

Figure 5.5: External instrumentation of a method

not directly available. In order to obtain the return value, one of a specific set of instructions has to be invoked, namely *move-result*, *move-result-object* or *move-result-wide*.

Once read the return value in one of the added registers, this, together with a reference of the *this* object are passed to the instrumentation hooks.

If the original bytecode tries to perform another *move-result* instruction after the injected instrumentation, this will result in an error because those instructions are only valid if placed right after an invocation. To this end the last instruction of the instrumentation has to be a specifically crafted invocation whose return value has to be the same of the one that has been read when performing the instrumentation. A solution to this problem is obtained using Java generics. In fact, for the return type of the crafted invocation to be whichever object type was the original return value, it is possible to use *Java generics* to force the return type to be the same as the original.

5.3 Instrumentation hooks

Instrumentation hooks are classes and methods specifically introduced to be easily referenced and invoked by the instrumented bytecode.

Those classes actually consists of a class used to store information, thus called *Payload*, and a set of other static methods that will be presented in this section.

5.3.1 The Payload class

The Payload class offers a set of methods to add parameters, setters in order to set the *API* identifier that has been triggered, a set of getters used to retrieve data from the Payload object, among which the eventual return value as

```

move-object vN+2, vThis
const-string vN+1, "API identifier"
new-instance vN, LInstrumented/hooks/Payload;
invoke-direct/range {vN, vN+1, vN+2}, LInstrumented/hooks/Payload;-><init>(Ljava/lang/String;Ljava/lang/Object;)V

move-object vN+1, vThis
const-string vN, "API identifier"
invoke-static {vN,vN+1}, LInstrumented/hooks/Payload;->createPayload(Ljava/lang/String;Ljava/lang/Object;)LInstrumented/hooks/Payload;
move-result-object v2

```

Figure 5.6: Registers usage comparison

described in Subsection 5.2.3, and a *log* method that performs the notification through logging.

The class does not offer public constructors, instances can instead be generated with a factory design pattern, this choice has been pondered in order to minimise the usage of registers in the injected bytecode (Figure 5.6). Two instance generators are available, one specifically created for instrumentation of method parameters at method start, and one for the return value. Both of them receive two parameters, this is because the number of added registers is two.

The first instance generator receives the reference to *this* object and the *API* identifier, and then adds each of the method parameters, whilst the second receives the *this* reference and the return value and requires the *API* identifier to be set afterwards.

An example of bytecode instrumented at the beginning of a method is shown in Figure 5.7. It is possible to notice the preamble of register shifting, followed by the the creation of the *Payload* object, with the *API* identifier and a zero value which represents *null* as reference to *this* object since the method is static. A series of *pushParam* follows, one for each parameter of the method, preceded by an eventual conversion to object of native types. Finally there is the invocation of the *log*, what happens during the invocation is described in the next subsections.

5.3.2 Call stack inspection

Once the injected bytecode has properly set the *Payload* containing all the needed information and its *log* method has been invoked, call stack inspection can be enabled in order to filter out legitimate invocations originated from the framework itself from the ones originated by other applications.

```

[1631ec] libcore.io.IoBridge.sendto:(Ljava/io/FileDescriptor;Ljava/nio/ByteBuffer;ILjava/net/InetAddress;I)I
0000: move-object v9, v11
0001: move-object v10, v12
0002: move v11, v13
0003: move-object v12, v14
0004: move v13, v15
0005: const/4 v14, #int 0 // #0
0006: const-string v15, "Llibcore/io/IoBridge; ->sendto(Ljava/io/FileDescriptor;Ljava/nio/ByteBuffer;ILjava/net/InetAddress;I)I"
0008: invoke-static {v14, v15}, Lit/polimi/elet/necst/droidsage/Payload;.newCallPayload:
(Ljava/lang/Object;Ljava/lang/String;)Lit/polimi/elet/necst/droidsage/Payload;
000b: move-result-object v14
000c: move-object v15, v9
000d: invoke-virtual {v14, v15}, Lit/polimi/elet/necst/droidsage/Payload;.pushParam:(Ljava/lang/Object;)V
0010: move-object v15, v10
0011: invoke-virtual {v14, v15}, Lit/polimi/elet/necst/droidsage/Payload;.pushParam:(Ljava/lang/Object;)V
0014: invoke-static {v11}, Ljava/lang/Integer;.valueOf:(I)Ljava/lang/Integer; // method@0874
0017: move-result-object v15
0018: invoke-virtual {v14, v15}, Lit/polimi/elet/necst/droidsage/Payload;.pushParam:(Ljava/lang/Object;)V
001b: move-object v15, v12
001c: invoke-virtual {v14, v15}, Lit/polimi/elet/necst/droidsage/Payload;.pushParam:(Ljava/lang/Object;)V
001f: invoke-static {v13}, Ljava/lang/Integer;.valueOf:(I)Ljava/lang/Integer;
0022: move-result-object v15
0023: invoke-virtual {v14, v15}, Lit/polimi/elet/necst/droidsage/Payload;.pushParam:(Ljava/lang/Object;)V
0026: invoke-virtual {v14}, Lit/polimi/elet/necst/droidsage/Payload;.log:()V

```

Figure 5.7: Example of instrumented bytecode

```

Number of iterations: 100000
Simple empty loop took 12 ms
Getting Thread stack trace took 24410 ms
Getting Throwable stack trace took 14677 ms

```

Figure 5.8: Stack trace acquisition

In order to perform call stack inspection the Android operating system offers two possible solutions through *Java APIs*. A first proposed solution takes advantage of the proper *getStackTrace* method in the class *Thread*, but this approach is known to be slower; an optimised alternative leverages the exception handler instantiating a *Throwable* object and using its *getStackTrace* method. The final result is the same, just the second approach is more efficient, as also proven by empirical tests run on Android 2.3 GingerBread (Figure 5.8).

The call stack, composed of an array of *StackTraceElement* objects is then analysed using a set of white-lists containing the most used package names offered by Java and Android, and the package names referred belonging to the instrumentation code itself (Figure 5.9).

If an entry in the call stack does not match any of package names the white-lists, the *API* called that triggered the instrumentation is marked as originated by an external application.

The white-list is saved in an *HashMap* allowing constant time access and thus improving the overall performances.


```
android.*
java.*
libcore.*
dalvik.*
com.android.internal.*
com.android.server.*
com.android.providers.*
org.apache.harmony.luni.platform.*
org.apache.harmony.luni.net.*
com.android.defcontainer.*
com.android.commands.monkey.*
org.objenesis.*
org.objectweb.asm.*
flexjson.*
com.fasterxml.*
```

Figure 5.9: List of white-listed package names

5.3.3 Serialisation of objects to JSON

Serialisation of objects to *JSON* is optionally used to obtain a more meaningful output. This feature is enabled switching the appropriate flag at runtime. Being data-centric, *JSON* typically improves the readability compared to alternatives such as *XML*.

Multiple serialisation software libraries have been tested:

GSON ¹This a software library developed by Google, the tested version was 2.2.3, GSON was found not able to correctly handle objects with circular references, this happens when structures such as doubly linked list are used. This is a known bug of the project, reported as Issue 137² on the project bug tracker.

FlexJSON ³FlexJSON is another software library to convert objects in JSON and vice versa, this project has the property of being very lightweight in terms of code base but still effective when compared to its competitors. FlexJSON performs deep serialisation of an object, and its computation time explodes when serialising highly nested objects resulting in stack overflow or memory exhaustion. The used version is 2.1 which was released in 2010.

¹<https://code.google.com/p/google-gson/>

²<https://code.google.com/p/google-gson/issues/detail?id=137>

³<http://flexjson.sourceforge.net/>

JackSON ⁴JackSON is considered more advanced compared to the previously described projects, this is because it allows further customisation of the serialisation process with tens of available plugins. Even though more customisable, similarly to FlexJSON it easily resulted in stack overflow exceptions and memory exhaustion when serialising highly nested objects.

While the idea to use GSON has been abandoned, FlexJSON and JackSON have been kept and are actually used one as fallback for the other.

JackSON has been further patched to reduce stack overflow errors and memory exhaustion, this was achieved introducing *depth awareness* in the serialiser, which means that the serialiser will stop serialising internal objects when a given depth (currently set as three) is reached (Figure 5.10).

Even solving stack overflow errors, in multiple occasions the serialisation has been found to be very slow, this was due to object containing high numbers of fields, an example is the class *android.view.View*.

To further limit time spent serialising objects, when the number of fields in a class is higher than a customisable number (currently fifteen), static fields are not serialised.

Another improvement was to avoid serialisation of fields inherited from classes whose distance, in terms of hops for the relation *child-of* from a subclass to a superclass, is higher than a given number (currently two), this filtering is only applied when the number of fields is higher than a custom number (currently twenty).

5.3.4 Logging of APIs

The logging functionality offered by this work may include, if enabled, call stack filtering and serialisation of objects to *JSON*, as described in previous sections, furthermore it is also possible to enable or disable the whole logging functionality. All these features can be enabled or disabled at runtime using a configuration file (Figure 5.11).

The logging methods are invoked by the *log* method in the *Payload* class, and they are *log_call* and *log_return* depending if the target of instrumentation were the API parameters or return value respectively.

The log entry is composed of a *JSON* object, whose fields are:

⁴<http://jackson.codehaus.org/>

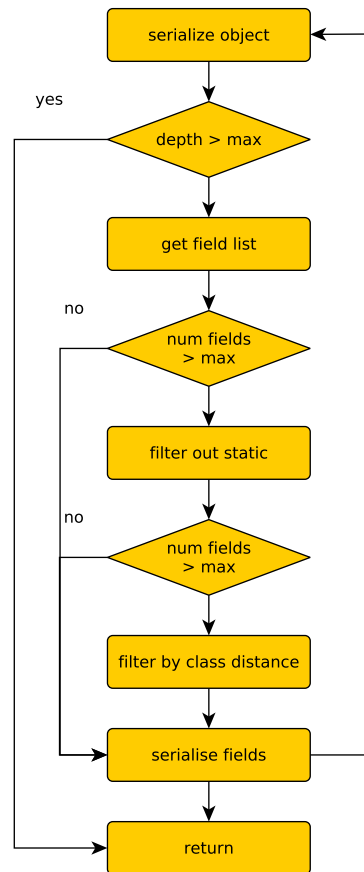


Figure 5.10: Depth awareness and field filtering in objects serialization

```
serialize=false  
stacktrace=true  
whitelist=org.apache.harmony.luni.platform  
whitelist=com.android.defcontainer  
whitelist=it.polimi.elet.necst.appfuzzer  
whitelist=com.android.commands.monkey  
whitelist=org.apache.harmony.luni.net
```

Figure 5.11: Runtime configuration file example

API signature This is a string representing the traced *API* who generated the log entry, it is obtained from the API identifier described in Section 5.1.

Callee object The serialisation of the object on which the *API* has been invoked, this is obtained from the reference to *this* object if the API is not a static method.

Parameters This is a JSON list containing the serialisation of the parameters used to invoke the API, it is only present in *log_call* method where the target of the instrumentation are parameters.

Return value Obtained as the JSON serialisation of the return value, this is only present in *log_return* method.

Call stack Obtained as described in Subsection 5.3.2, it is a list of *StackTraceElement* each of them indicating class and method name of the invoked method in the call stack.

The composed string is then logged using the Android logging functionality *android.util.Log*. The reasons behind this choice is ease of use and the fact that, differently from using network sockets, it does not require specific permissions to produce log entries.

The default android logging class, has been found having issues with long texts, specifically when the length of text and log header is higher than 4096 characters, this has been circumvented invoking the logging APIs once every 4096 characters of output.

Since this small fix may introduce thread synchronization issues, causing logging entries to be potentially mixed, both *log_call* and *log_return* will invoke a single method set as statically synchronized. This should ensure a thread synchronization when generating the log as output.

Some examples of output logs are show in Figure 5.12 and Figure 5.13, the first with serialisation of parameters and callee object enabled and the second with it disabled respectively.

5.4 System image generation

This section describes how the system image, which will be used as black-box dynamic analysis platform, is created starting from an existing user supplied

```

I/DRSAGE ( 2223): { "api_sign" : Ldalvik/system/DexClassLoader;-><init>(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;Ljava/lang/ClassLoader;)V, "caller" : {"parent":null}, "stack_trace" : [ "com.facebook.base.app.DelegatingApplication->attachBaseContext", "com.facebook.base.app.DelegatingApplication->f", "com.facebook.katana.app.FacebookApplication->a", "com.facebook.katana.app.FacebookApplication->f", "com.facebook.common.dextricks.DexLibLoader->ensureDexsLoaded", "com.facebook.common.dextricks.DexLibLoader->c", "com.facebook.common.dextricks.DexLibLoader->a", "com.facebook.common.dextricks.DexLibLoader->a" ], "params" : [ ["/data/data/com.facebook.katana/app_secondary_program_dex/program-8074103ff6245f68be39e8b1b5401f1d08ec3707f1783aa9.dex.jar","/data/data/com.facebook.katana/app_secondary_program_dex_opt",null,{"parent":{"parent"}}] ] }

I/DRSAGE ( 1618): { "api_sign" : Llibcore/io/0s;->connect(Ljava/io/FileDescriptor;Ljava/net/InetAddress;I)V, "caller" : {"egid":10021,"euid":10021,"gid":10021,"pid":1618,"ppid":85,"uid":10021}, "stack_trace" : [ "com.google.android.gms.playlog.uploader.UploaderIntentService->onHandleIntent", "dbe->e", "dbe->a", "dbe->a", "fni->execute", "fni->execute", "fni->a", "fnd->execute", "org.apache.http.impl.client.AbstractHttpClient->execute", "org.apache.http.impl.client.AbstractHttpClient->execute", "org.apache.http.impl.client.DefaultRequestDirector->execute", "org.apache.http.impl.conn.AbstractPooledConnAdapter->open", "org.apache.http.impl.conn.AbstractPoolEntry->open", "org.apache.http.impl.conn.DefaultClientConnectionOperator->openConnection", "org.apache.http.conn.scheme.PlainSocketFactory->connectSocket" ], "params" : [ [{"int$":47},"173.194.70.95",443] ] }

```

Figure 5.12: Example of logs with serialisation

```

I/DRSAGE ( 1510): { "api_sign" : Llibcore/io/0s;->connect(Ljava/io/FileDescriptor;Ljava/net/InetAddress;I)V, "caller" : "libcore.io.Posix@410573a0", "stack_trace" : [ "com.lookout.security.InstallReceiverService->onHandleIntent", "com.lookout.c.c.c.a->a", "com.lookout.c.c.c.a->a", "com.lookout.c.c.n->a", "com.lookout.c.d.b->a", "com.lookout.c.d.b->a", "com.lookout.c.c.f->a", "com.lookout.n.k->a", "com.lookout.security.j->a", "com.lookout.s.k->a", "com.lookout.utils.HttpUtils->executeWithAuth", "org.apache.http.impl.client.AbstractHttpClient->execute", "org.apache.http.impl.client.AbstractHttpClient->execute", "org.apache.http.impl.client.DefaultRequestDirector->execute", "org.apache.http.impl.conn.AbstractPooledConnAdapter->open", "org.apache.http.impl.conn.AbstractPoolEntry->open", "org.apache.http.impl.conn.DefaultClientConnectionOperator->openConnection", "org.apache.http.conn.scheme.PlainSocketFactory->connectSocket" ], "params" : [ "[FileDescriptor[72],/184.169.142.1" ] ] }
I/DRSAGE ( 1510): 40, 443] ] }

I/DRSAGE ( 1618): { "api_sign" : Landroid/content/ContextWrapper;->startService(Landroid/content/Intent;)Landroid/content/ComponentName;;, "caller" : "com.google.android.gms.common.app.GmsApplication@413daca0", "stack_trace" : [ "com.google.android.gms.gcm.PushMessagingRegistrarProxy->onHandleIntent" ], "params" : [ "[Intent { act=com.google.android.c2dm.intent.REGISTER cmp=com.google.android.gms.gcm.GcmRegisterService (has extras) }]" ] ] }

```

Figure 5.13: Example of logs without serialisation

and thus fully customisable system image.

5.4.1 Image extraction

The given system image has to be extracted in order to access the files it comes with. This is required since Android disk partitions normally use YAFFS2⁵ as file system. This kind of file system is optimised to work with block devices such as *NAND* chips typically used for smartphones internal *non-volatile* memory.

The only software found to perform the extraction are *unyaffs*⁶ and *unyaffs2*⁷. These two tools parse the raw image of the YAFFS file system and recreate every file and folder it contains in another destination file system supported by the host. While both software have been found working, DROIDSAGE currently uses *unyaffs*.

Particular attention has to be made to preserve file attributes such as owner and group, the extracted files will in fact may have owner and group of the user who performed the extraction. This problem is addressed in DROIDSAGE when rebuilding the final instrumented image.

5.4.2 Dalvik bytecode identification

Once extracted the user-supplied image as described in the previous Subsection 5.4.1 all the bytecode has to be located in order to be instrumented where necessary.

A simple yet effective approach consists in walking the folder of the extracted image in order to locate *APK* (Android PacKage) files, *JAR* (Java ARchive) files, *DEX* files and *ODEX* files.

Depending on their extension, the files are further processed:

JAR These files in an Android system image might contain Dalvik bytecode depending whether it has been optimised or not. If the JAR file has not been optimised, a *classes.dex* file is present containing the bytecode, otherwise the bytecode is removed from the archive and placed in a separate ODEX file. Thus if a *classes.dex* file is present, it is extracted and further processed.

⁵<http://www.yaffs.net/>

⁶<https://code.google.com/p/unyaffs/>

⁷<https://code.google.com/p/yaffs2utils/>

APK Similarly to what happens with JAR archives, APK files may contain Dalvik bytecode in the case they have not been optimised.

DEX Dalvik executable files contain various sections as described in Section 3.3, one of which lists all the classes defined in the file. For each class in this section is then defined the list of attributes, fields and methods it has. Finally for each method defined in a class the bytecode representing its instructions is present, as long as it is not an abstract or native method. Abstract methods are present in abstract classes and interfaces which are typical in Java language, while native methods are methods that can be invoked as regular methods, but whose definition is not in the form of Dalvik bytecode. Native methods are in fact defined in *ELF* binaries, normally obtained as a result of compiled *C* or *C++* code, these binaries interact with the Dalvik virtual machine using *JNI* (Java Native Interface). These native methods as described in Subsection 5.2.3 are not directly instrumentable by our system whose target is only Dalvik bytecode.

ODEX ODEX files can be considered as an extension of DEX files, they in fact embed the structure of a DEX file internally. ODEX files while being optimised in terms of instructions over time, they are not optimised in terms of space requirements, they in fact add other contents to the one of the originating DEX.

5.4.3 Conversion of ODEX into DEX

This step is necessary to keep the work-flow more generic, to allow further customisation of the approach and to simplify the rest of the process.

The one and only known software tool able to convert ODEX into DEX is BAKSMALI⁸. Every other software claiming to perform conversion of ODEX files to DEX was actually found to be using BAKSMALI. This software tool disassembles Dalvik bytecode in a text representation that can be manually modified and then rebuilt. Multiple attempts in taking advantage of BAKSMALI have been performed, but several crashes of the software forced looking for alternative solutions. The version used was the latest available when the attempts were made, namely version 1.4.2. Another reason to prefer avoiding

⁸<https://code.google.com/p/smali/>

Algorithm 5.1 Division in basic blocks

```

create basic block at 0
for each instr:
  currentBB = getBasicBlock( instr )
  if jumps( instr ):
    afterBB = create new basic block after instr;
    dest = destination( instr )
    oldBB = getBasicBlock( dest )
    if not exists( basic block starting at dest ):
      destBB = create new basic block at dest;
      if backwardJump( instr ):
        link( oldBB -> destBB )
    link( currentBB -> destBB )
    if conditionalJump( instr ):
      link( currentBB -> afterBB )
  if terminates( instr ):
    afterBB = create new basic block after instr;

for each tryCatch:
  tryStart = getTryStart( tryCatch )
  tryEnd = getTryEnd( tryCatch )
  tryBlocks = getBasicBlocksInRange( tryStart, tryEnd )
  catchHandlers = getCatchHandlers( tryCatch )
  for each tryBlock:
    for each catchHandler:
      handlerAddr = getHandlerAddress( catchHandler )
      handlerBB = getBasicBlock( handlerAddr )
      link( tryBlock -> catchHandler )

```

the usage of BAKSMALI is the fact that when rebuilding disassembled bytecode some optimisation is applied. This causes an alteration of the bytecode that will then differ respect to the original even when no instrumentation is performed as described in the following.

To solve this issue our system introduces extra steps when parsing the bytecode. The content of an ODEX file is analysed and accordingly to the details reported in Section 3.4 the embedded DEX is located and processed.

The process first locates each method definition in the embedded DEX file, then the bytecode of each method is parsed in three steps:

Division in basic blocks A first step is necessary in order to determine the basic blocks present in the bytecode (Algorithm 5.1). This is achieved creating a new basic block after every conditional or unconditional jump, to this end instructions such as *return* and *throw* will be considered unconditional jump instructions. The target of the jump will also determine the start of another basic block. Lastly, each catch block present in the bytecode also starts a new basic block .

During this phase a bidirectional graph representing the *CFG* (*control flow graph*) is built accordingly to basic blocks reachability[Theiling(2000)];


```

try{
    int v = in.read();
    if( v != 0 )
        v = in.read();
    in.close();
}catch(IOException ioe){
    // do nothing
}

```

```

|0000: invoke-virtual {v1}, Ljava/io/InputStream;.read:()I
|0003: move-result v0
|0004: if-eqz v0, 0009 // +0005
|0006: invoke-virtual {v1}, Ljava/io/InputStream;.read:()I
|0009: invoke-virtual {v1}, Ljava/io/InputStream;.close:()V
|000c: return-void
|000d: move-exception v0
|000e: goto 000c // -0002

catches      : 1
0x0000 - 0x000c    Ljava/io/IOException; -> 0x000d

```

Figure 5.14: Structure of a try-catch in Dalvik bytecode

A catch block is considered reachable from all of the basic blocks in the try block.

Another important phase of this step is keeping track of register assignments, more precisely keeping track for every assignment the object types a register can hold. This assignment can be fixed or inferred. Fixed assignments are given by fixed type instructions, such as arithmetic operation instructions, new instance generation instruction or constant values instructions. Inferred assignments depend on the value of other registers. These are kept unresolved during this step since a global view of the *control flow graph* is not yet available.

Register type propagation According to the *CFG*, a map of the types of objects each register can assume is created [Cytron et al.(1991)]. This does not require the bytecode to be parsed again, but it requires knowledge of the *control flow graph* and register assignments, thus can be computed only after the previous step.

Each basic block inherits object types from its predecessors in the *control flow graph*, except for the first basic block whose inherited register types are considered to be the method parameters and the reference to *this* in the case of non-static methods.

The map of registers to object types is built using the register assignments traced in the first step. In particular, register types have to be computed for the end of each basic block and these are added as inherited types for the next basic blocks in the *control flow graph*.

When multiple assignments are performed to the same register in a basic block, only the last one is propagated as inherited register type.

As shown in Figure 5.14 special attention has to be taken for try-catch

blocks since a catch block could be reached from each instruction in the try blocks. To this end, inherited register types have to be computed differently for each block belonging to the address range of the try.

For the first block in the try, register types are computed at the beginning of the try (possibly after the start of the block itself), then for each register assignment until the end of the try or the block, its value is added to the inherited types. This is repeated for the rest of the blocks in the try, starting from their beginning.

The algorithm used for register type propagation is a breadth first exploration of the basic blocks, with the exception of catch blocks, that will be processed only after all the basic blocks in the correspondent try have been processed.

Since inherited types can assume multiple values depending on the basic block it has been inherited from, values are automatically updated when new values are found.

De-optimisation The bytecode is then parsed a second time, this time regular DEX bytecode can be generated as output replacing optimised instructions with their un-optimised equivalent, as specified in Table 5.1 . There have been a few cases in which it was not possible to substitute optimised instruction. After manual inspection of more than twenty instances, this has always been caused by dead code, especially duplicated code in *finally* statements, for which no valid register types were found. In those cases bytecode instructions that will raise an exception are injected. This allows to easily track any error in the case it was not dead code. As a matter of fact, no exceptions have been raised by the injected instructions in the tests performed, which empirically confirm it was dead code.

5.4.4 Class graph and call graph generation

Once converted the ODEX files to regular DEX files, these and the other DEX files in the system image are analysed with the goal of creating a global call graph and class graph of all the bytecode in the image itself.

To this end a relational database will be created and used: for ease of use a

Optimised instruction	Un-optimised equivalent	Description
invoke-virtual-quick	invoke-virtual	Invokes a virtual method
invoke-super-quick	invoke-super	Invokes a virtual method of the parent class
invoke-direct-empty	invoke-direct	Invokes a direct method
execute-inline	invoke-(direct virtual static)	Invokes a method in a set of predefined ones
iget-quick	iget(-(boolean byte char short))?	Gets a native type from an object field
iget-object-quick	iget-object	Gets an object type from an object field
iget-wide-quick	iget-wide	Gets a native wide type from an object field
iget-volatile	iget(-(boolean byte char short))?	Gets a native type from a volatile object field
iget-object-volatile	iget-object	Gets an object type from a volatile object field
iget-wide-volatile	iget-wide	Gets a native wide type from a volatile object field
iput-quick	iput(-(boolean byte char short))?	Sets an object field from a native type
iput-object-quick	iput-object	Sets an object field from an object type
iput-wide-quick	iput-wide	Sets an object field from a native wide type
iput-volatile	iput(-(boolean byte char short))?	Sets a volatile object field from a native type
iput-object-volatile	iput-object	Sets a volatile object field from an object type
iput-wide-volatile	iput-wide	Sets a volatile object field from a native wide type
sget-volatile	sget(-(boolean byte char short))?	Gets a native type from a static field
sget-object-volatile	sget-object	Gets an object type from a static field
sget-wide-volatile	sget-wide	Gets a native wide type from a static field
sput-volatile	sput(-(boolean byte char short))?	Sets a static volatile field from a native type
sput-object-volatile	sput-object	Sets a static volatile field from an object type
sput-wide-volatile	sput-wide	Sets a static volatile field from a native wide type
return-void-barrier	return-void	Similar to return-void, specific for Symmetric MultiProcessing

Table 5.1: List of optimised instructions

SQLite database has been chosen. The schema of this database is represented in Figure 5.15.

The call graph is stored as a list of entries in the *invocations* table, each entry composed of a reference to the caller method and a reference to the callee method. The class graph is represented by the two tables *class* and *implements*, the first stores information regarding a class, while the second is used to track interfaces that a class implements.

Each method has also information regarding the class in which it is defined and attributes to determine if it is directly instrumentable or not, namely if it is not abstract or native as described in Subsection 5.4.6.

The two graphs are created parsing the bytecode of the DEX files and looking for class definitions, method definitions and method invocation instructions inside each method (see Appendix A), for each of those instructions an entry is inserted in the call graph.

Information stored in the database will be used in the next steps to detect when and where to instrument a given method.

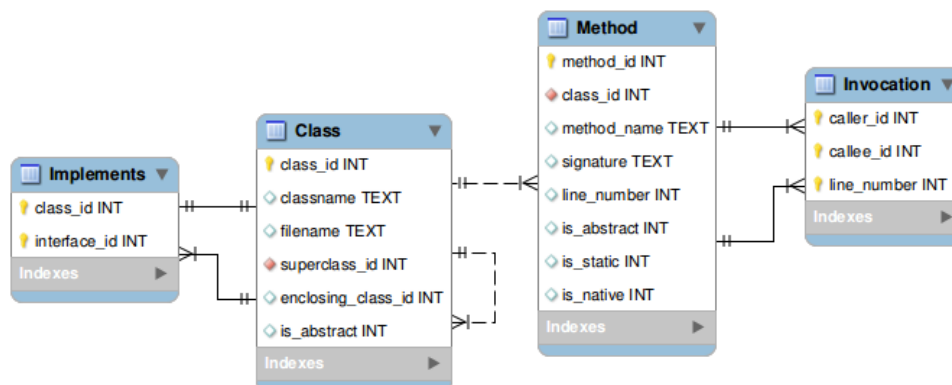


Figure 5.15: Class and call graph database schema

5.4.5 User decision of target APIs

Both class graph and call graph obtained in the previous step could be used to determine which API to trace and thus to instrument, this is particularly useful when the system image that is going to be instrumented is not well known to who has to decide the APIs to trace.

To this end, this work includes a graphical representation in *HTML* and *AJAX* that offers a navigable graph of classes and methods, easy to use for an end user in order to identify the most suitable APIs to be instrumented. A screenshot picturing this graphic tool is shown in Figure 5.16.

Once decided the APIs to be traced, these have to be written in a *CSV* (*comma separated values*) file containing for each API the class name and method name it refers to. It is possible to further specify a method description with parameters and return type of the method to resolve ambiguous situations. Lastly it is also possible to specify whether to track the method when it is going to be invoked with its parameters or to track it when it is about to return to the caller, thus with its return value. This last customisation is particularly useful when instrumenting methods which perform operations such as reading from a stream, in those cases is typically more significant to track their return value rather than the method parameters.

An example configuration file is shown in Table 5.2 .

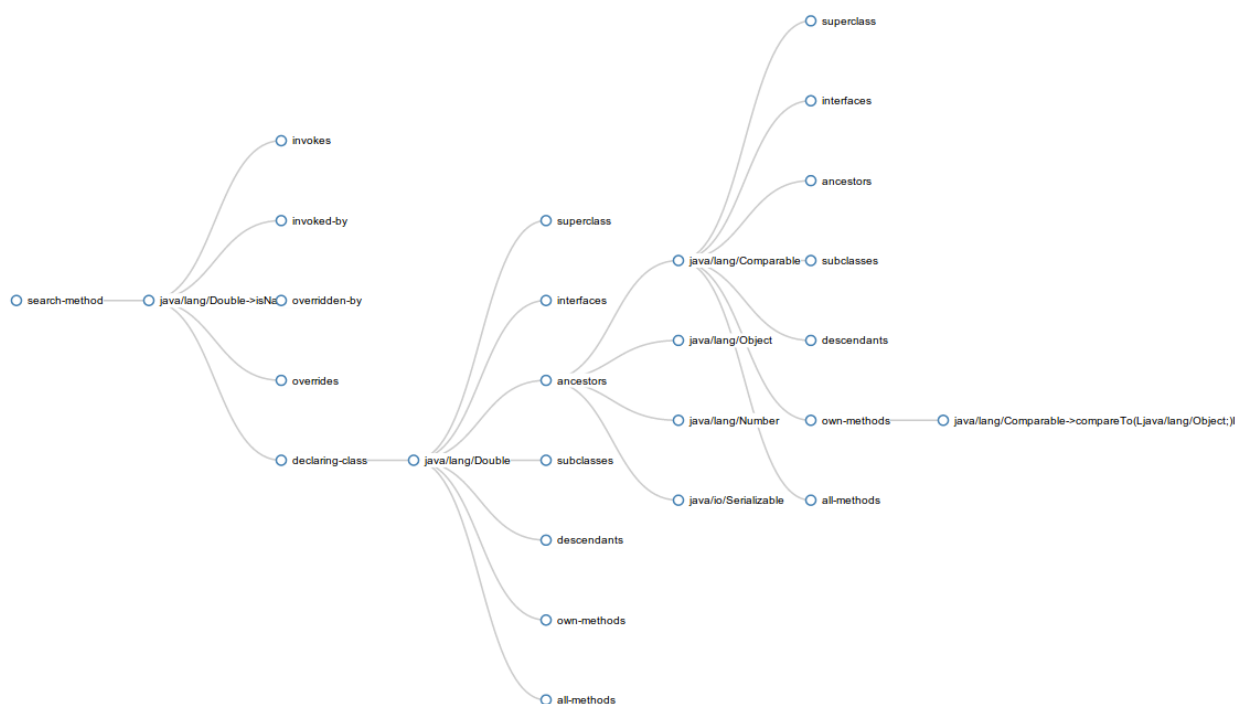


Figure 5.16: Navigable graph of classes and methods

API signature	log call	log return
Ljavax/crypto/Cipher;->doFinal ([B] [B	true	false
Landroid/app/Activity;->startActivity	true	false
Landroid/telephony/SmsManager;->sendTextMessage	true	false
Landroid/telephony/TelephonyManager;->getDeviceId	true	false

Table 5.2: Example of configuration file listing the APIs to instrument

5.4.6 Instrumentation

The analysed DEX files are instrumented on the basis of the call graph generated in Subsection 5.4.4, for each *DEX* file its bytecode is parsed and for each method it is detected whether to instrument its bytecode or not.

The bytecode of a method has to be instrumented in the case it appears in the target *API* list or if it contains an invocation of a non-directly instrumentable method which is also listed as an *API* to track.

Details on how the instrumentation is performed are available in Section 5.1.

```
$ dexopt
Usage:
```

```
Short version: Don't use this.
```

```
Slightly longer version: This system-internal tool is used to
produce optimized dex files. See the source code for details.
```

Figure 5.17: Error message when running *dexopt*

5.4.7 Bytecode optimisation

Once the bytecode has been instrumented, it is optimised into ODEX format. This has the main advantage of allowing the modified bytecode to be run without performing the signature verification that if otherwise performed would result in a wrong signature.

The optimisation process uses the *dexopt* binary which is the official tool to generate ODEX bytecode.

Dexopt is available in two versions, the host and the target version. The target version is the one included in the system image that is directly run from a device running the Android operating system, the host version is instead used when building a system image from source code, in these cases a pre-optimisation can be performed from the host system that is building the image. The two versions are basically the same binary cross-compiled for different platforms and architectures.

Dexopt is meant to be automatically invoked, either by the *make* command when building a new image, or directly from Android when installing a new application, it does not offer in fact a simple interface to invoke it manually as shown in Figure 5.17, it has in fact been necessary to inspect its source code to understand its configuration and its available parameters.

When *dexopt* fails an optimisation it often requires manual and in depth analysis of both the binary and the bytecode, normally using hexadecimal editors, to find the reason behind the failure.

5.4.8 Image creation

As a last step, the instrumented system image is built using *mkyaffs2image*, an official tool used when building an Android system image. As described in Subsection 5.4.1 it is necessary to preserve file attributes such as owners and

groups, to this end *mkyaffs2image* offers a “-f” command line switch to set file owner and group appropriately.

The resulting image can either be run in an emulator or installed on a physical device.

Chapter 6

Experimental validation

The realised system has been tested with multiple configurations that will be described in this chapter.

First of all, in order to compare different configurations, several system images have been prepared. Each system image differs from the others at least in one characteristic, such as Android version or installed applications, allowing us to estimate which factors most infer on test results.

The testing process then involves samples of Android applications, which have been chosen including malware as well as known good applications. Each of these samples is installed in clean Android platforms that run instances of the prepared system images.

Once installed, each application sample gets stimulated through the Android Debug Bridge, this includes starting each of the activities, services and broadcast receivers that the application contains. Each activity is further stimulated using the MONKEY tool that sends events to the Android system. MONKEY has been configured to run predefined scripts, this helps highlighting differences in application behaviours which might be due to the underlying environment in which the application is being run. If during the stimulation other applications are installed on the system, these are added to an application queue, so that when the test of the current application ends, another test of the newly installed one will be started on the same system. This last detail allows to correctly analyse even complex malware that ask the installation of a second application that contains malicious code. An example of this behaviour can be found in the well known Android trojan application

*SMSZombie*¹ that carries a hidden APK file disguised as a JPEG image.

Throughout all the stimulation phase, APIs invocations are logged. This allows to post-process them with the aim of comparing how the different configurations impact on applications' behaviour.

Behaviours are compared in terms of different APIs invoked and amount of invocations each of those APIs has reached in a test.

To compare our approach with existing ones, DroidBox has also been chosen as one of the system images on which to run the tests.

6.1 Testbed environments

Multiple Android platform versions have been chosen as environments on which to perform the tests. A single Android version could then be further distinguished by the applications it comes with. Particular interest may be given to the presence (or absence) of the *Google Apps*, in fact their presence could enable an application to interact with services such as *Google Maps*, *Google Plus* or *Youtube* which may enable (or disable) specific behaviours in the application.

In this section a list of the prepared system images will be introduced, then details about the stimulation of applications will be given and an automated testing framework will be presented.

6.1.1 System images

DROIDBOX has been chosen as the system to compare with since it can be considered the most known and currently widespread Android dynamic analysis sandbox platform.

As described in Subsection 2.2.2 the system image used by DROIDBOX is based on TAINTDROID, and has added instrumentation for specific APIs. The instrumentation of those APIs is mainly aimed at logging the usage of given methods, but the logic behind whether to log or not an API invocation could be complex.

By manually inspecting the source code of DROIDBOX a set of APIs that always perform logging has been identified. Those APIs have been chosen to be instrumented with our system on different Android platforms.

¹<http://blog.fortinet.com/uninstallable-androidsmszombie/>

Algorithm 6.1 List of APIs instrumented for Android 2.3 GingerBread

```

Ldalvik/system/DexClassLoader;
-><init>(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;Ljava/lang/ClassLoader;)
Ljavax/crypto/spec/SecretKeySpec;
-><init>
Landroid/content/ContextWrapper;
->startService(Landroid/content/Intent;)
Lorg/apache/harmony/luni/platform/OSNetworkSystem;
->connect(Ljava/io/FileDescriptor;Ljava/net/InetAddress;II)
->connectNonBlocking(Ljava/io/FileDescriptor;Ljava/net/InetAddress;I)
->close(Ljava/io/FileDescriptor;)
->send(Ljava/io/FileDescriptor;[BIIILjava/net/InetAddress;)
->sendUrgentData(Ljava/io/FileDescriptor;B)
->write(Ljava/io/FileDescriptor;[BII)
->readDirect
->recvDirect
Landroid/telephony/SmsManager;
->sendTextMessage

```

Given that TAINTDROID is based on Android 2.3 GingerBread, an instrumented image has been generated from the stock Android 2.3 GingerBread system image shipped with the Android SDK.

In order to see how different behaviours happen depending on the environment, other two system images have been prepared: one based on Android 4.0 IceCreamSandwich, and one based on Android 2.3 GingerBread, but with the entire Google Service Framework installed.

Here follows a list of the prepared system images:

Baseline (DroidBox 2.3) This system image is taken from the DROIDBOX project ²and can be freely downloaded from the official download page³. The image has been used as is without further customisation.

Our system (GingerBread 2.3) The stock Android 2.3 Gingerbread system image from the Android SDK has been instrumented with our system. The list of instrumented APIs is shown in Algorithm 6.1.

Our system (GingerBread 2.3 + GSF) The original GingerBread 2.3 system image has been first modified including the Google Service Framework, obtained from *Goo.im*⁴. The resulting system image has then been instrumented with our system. The list of APIs instrumented coincides with the ones shown in Algorithm 6.1 for the previous system image.

²<https://code.google.com/p/droidbox/>

³<https://droidbox.googlecode.com/files/DroidBox23.tar.gz>

⁴<http://goo.im/gapps>

Algorithm 6.2 List of APIs instrumented for Android 4.0 ICS

```

Ldalvik/system/DexClassLoader;
-><<init>(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;Ljava/lang/ClassLoader;)
Ljavax/crypto/spec/SecretKeySpec;
-><<init>
Landroid/content/ContextWrapper;
->startService(Landroid/content/Intent;)
Llibcore/io/IoBridge;
->connect(Ljava/io/FileDescriptor;Ljava/net/InetAddress;II)
->closeSocket(Ljava/io/FileDescriptor;)
->sendto(Ljava/io/FileDescriptor;Ljava/nio/ByteBuffer;II;Ljava/net/InetAddress;I)
->recvFromBytes(Ljava/io/FileDescriptor;Ljava/lang/Object;II;Ljava/net/InetSocketAddress;)
Landroid/telephony/SmsManager;
->sendTextMessage

```

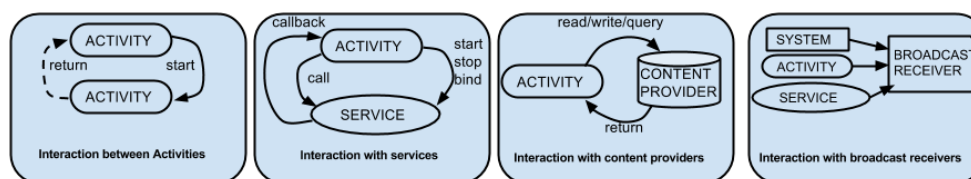


Figure 6.1: Interaction between Android components

Our system (IceCreamSandwich 4.0) This version is obtained from the original Android 4.0 Ice Cream Sandwich (ICS) system image present in the Android SDK. Not all the APIs that DroidBox references are available in this system image since the underlying platform differs, thus custom instrumentation has been performed. The instrumented APIs are shown in Figure 6.2.

6.1.2 Stimulation of a single application

Although application stimulation is not the primary goal of our approach, we needed a mechanism to streamline the experiments in a repeatable manner. For this, we adopted the approach described in the following.

Android applications have common components that will be invoked by the system when dispatching Intents. These components, whose interaction is shown in Figure 6.1, are typically divided in four main categories⁵:

Activities An activity represents a single screen with a user interface. Although the activities work together to form a cohesive user experience in an application, each one is independent of the others. In fact it may be possible for an Activity to be started from an external application.

⁵<https://developer.android.com/guide/components/fundamentals.html>

Services Services run in the background typically performing long-running tasks or interacting with input/output such as network connections or files. Differently from Activities, a Service does not have any graphical part.

Receivers A broadcast receiver is a component that responds to system-wide broadcast announcements. Intents carrying those announcements can both be started by the application itself or from other applications. Some of the Intents are also broadcasted by the operating system, announcing global events.

Providers ContentProviders allow storing data that has to be persisted. Stored data can also be accessed as well as modified, accordingly with the implementation of the ContentProvider. This kind of component can both be kept private or exposed to other applications depending on the configuration of the ContentProvider.

The invocation of these components can be forced sending custom Intents, but this requires root access to the Android system. Intent generation happens by means of ADB and the AM (Activity Manager)⁶ component which is available in Android system images. The realised system images have been prepared with root access enabled in order to allow forcing the invocation of the available components. This process, even if sometime causes the application to crash, normally allows to significantly increase code coverage of the tested application. When an activity under test is successfully displayed, the input generation tool MONKEY⁷ is started. MONKEY has been configured so that it will generate and execute a predefined list of events, with a fixed timing between them. This is particularly useful to detect different behaviours that appear when running the same activity, of the same application, with the same input. Any eventual difference in this scenario has a good chance of being due to the environment setup.

The events that MONKEY generates may cause the application to exit from the test scope, this typically happens when the *home* button is pressed, or when a series of back button is executed. In the first case, the home launcher activity is displayed, while in the second case the activity that was executing before the test was started is displayed.

⁶<https://developer.android.com/tools/help/adb.html#am>

⁷<https://developer.android.com/tools/help/monkey.html>

To address the first problem, a custom home launcher application has been developed so that every time the custom home screen is reached, a log entry is generated. This log entry being processed allows to recognise what happened and restart the activity under test. An additional trick has to be used to properly handle home button events, that is setting the custom launcher application as the default one.

Back to the case of a series of back button events, to address it, the home screen is displayed before starting the test of an activity. This ensures the previous running activity is under our control.

The list of events that MONKEY will execute is generated by a custom *python* script. This script is able to generate either typing events or touch events.

Each of these events is composed of sub-events, and each sub-event is interleaved by a randomly chosen small amount of time.

Touch events are composed by at least a touch down event and a touch up event, additionally other touch move events can be randomly generated.

The randomness of these events is described by a Poisson process, generated using the python API `random.expovariate`. The mean values have been chosen empirically trying to reproduce a real user behaviour.

The list of activities, services and broadcast receivers to test in each application is obtained through static analysis on the application itself. In particular the `classes.dex` file containing the Dalvik bytecode part of the application is parsed and a class hierarchy graph is built. On the basis of this graph, all descendants of Activity, Service and BroadcastReceiver are located.

The obtained list is then used to fulfil the testing of the application, forcing to start each component in the list.

6.1.3 Automatic stimulation of multiple applications at once

The tests have been run on a high number of samples, this fact required to generate an automated analysis environment. Given a running Android platform, an application, and a list of MONKEY events, the automated environment automatically installs the application in the Android system, then starts its stimulation forcing each activity, service and broadcast receiver to start.

When an activity is started, the list of input events is executed using the monkey command.

Throughout all the test, a log is kept in order to allow postprocessing the collected data.

In order to ensure each test being run in a clean system, snapshots are created. As a matter of fact this also reduces the application start time, avoiding to boot a clean system each time. On the other hand, this restricts the testing scope to emulated environments only.

6.2 Dataset

The application samples the tests have been run with were obtained both from known good sources, and from known Android malware datasets. More detailed information about the application samples can be found in this chapter.

6.2.1 Normal applications

The top ten Android application from five categories of the Google Play Store⁸ have been downloaded on the 25th/09/2013. The chosen categories are: education, entertainment, game, lifestyle and weather.

An important fact has to be considered, that is not all applications were actually able to run on all the test environments, which were created on the system images used to run the tests. This is because some of the applications may require the underlying environment to have specific configurations that might include the Google Services Framework, other specific applications such as Google Maps installed, or simply because of incompatible Android APIs versions. Generally speaking, applications compatibility may be influenced also by the presence (or absence) of hardware components, such as sensors such as GPS or GSM. Specifically, in our dataset two applications were incompatible with Android 2.3, and three applications required the Google Services Framework.

6.2.2 Malware applications

With respect to Android malware, 20 samples were chosen in the ones belonging to the Malgenome project⁹, and these are listed in Table 6.1 .

⁸<https://play.google.com/store/apps>

⁹<http://www.malgenomeproject.org/>

Malware	MD5	Date
DroidKungFu3	ca4a8e620f5ed94abf8232c59404dead	2011-09-01
BaseBridge	eba8c1fb8ce65a4c738eb2e1fc00d0dc	2011-06-14
Geinimi	50d85dc895a9ffe66c809416ad2cba91	2010-09-25
PjApps	cae2169f4706e8be514d9967152590c4	2010-12-17
KMin	e0dd14720e4fef02d29a4b027536d273	2011-04-11
GoldDream	fb019dacc3631e64861c814f01584e	2011-09-12
DroidDreamLight	3ae28cbf5a92e8e7a06db4b9ab0a55ab	2011-04-15
ADRD	839c37f3a2c8d31561d28f619a2a712e	2011-01-04
jSMShider	cc2a186f466431570109251f1d46cc36	2009-06-10
Zsone	de7e432e5ac86c34f315695a34ab8b6b	2011-01-12
FakeLookOut	65baecf1fe1ec7b074a5255dc5014beb	2012-10-06
RootSmart	f70664bb0d45665e79ba9113c5e4d0f4	2011-10-09
Walkinwat	c3a0f5d584cc2c3221bbd79486578208	2011-02-02
zHash	5beae5543a3f085080c26de48a811da6	2011-01-21
FakePlayer	46a53f4a6637e2807d79102a6a937c2e	2010-10-12
Spitmo	cfa9edb8c9648ae2757a85e6066f6515	2011-07-22
HippySMS	1a4fb41b95c3cbab05636ee966043c9e	2010-09-15
LoveTrap	f3497516eab17c642c5ede5ad1e55a15	2011-06-20
SmsSend	1385a0ebd5865b3e20456bfd97ac3a17	2011-05-24
FakeAlert	084a7b576f5df438abba3131a90af493	2013-01-17

Table 6.1: Malware samples

The reasons that led to choose the Malgenome project were mainly due to the fact that it is the most known and acknowledged Android malware dataset in the scientific community [Zhou and Jiang(2012)].

6.3 Test description

Two types of tests have been run on the system images listed in Subsection 6.1.1, these will be described in the details in this section as well as their results.

6.3.1 No stimulation

A first test has been performed with no input events. This test included a simple MONKEY script with a *UserWait* action which leads the system to wait for a specified amount of time.

This test has been performed on all the system images described in Subsection 6.1.1.

The logs obtained from the tests have been post processed and for each logged API, its invocations on the different combinations of system images and applications have been isolated.

The idea is to show how many invocations of a single API a system intercept respect to other systems for a generic application. So we first group the logged API invocations by system, application and API name.

Starting from the number of invocations of an API in a system image running a given applications, a comparison between the amount of API invocations logged by the different systems has been computed.

In first place, a label has been applied to each of the logged APIs. This process has the goal of normalising the API name with the correspondent action to allow cross-system comparison. To give an example, the API logged by DROIDBOX as “openNet” is labelled as “socket-connect”, while the API that our system logs as “Ljavax/crypto/spec/SecretKeySpec;-><init>” is labelled as “crypto-key”.

For each couple of tested application and label assigned, the total of logged API invocations with that label is computed, as well as the relative amount for each system. The computed amount is a number between 0 and 1, indicating the ratio of labelled invocations that a given system has logged over the total amount of logged invocations for that label and application.

In the case of no logged API invocations with a given label for an application, the ratio assigned to each system is equal to one over the number of compared systems.

For each system, mean and standard deviation are then computed with respect to the described ratio across the different applications.

The computed results for each label give an idea of the effectiveness of a system in logging a behaviour when compared to the others, or more specifically, how many invocations of a single API a system intercepted respect to other systems for a generic application.

6.3.2 Stimulation with random events

A second test has been performed with a list of random input events. As described in Subsection 6.1.2, the same input was supplied to all tested activities on all the tested applications.

```

start data >>
DispatchPointer(0, 0, 0, 282, 421, 0,0,0,0,0,0,0)
DispatchPointer(20, 0, 1, 254, 467, 0,0,0,0,0,0,0)
DispatchPointer(0, 0, 0, 4, 89, 0,0,0,0,0,0,0)
DispatchPointer(0, 0, 2, 0, 20, 0,0,0,0,0,0,0)
.....
DispatchKey(20,0,0,62,0,0,0,0)
DispatchKey(20,0,1,62,0,0,0,0)
DispatchKey(20,0,0,39,0,0,0,0)
DispatchKey(20,0,1,39,0,0,0,0)
DispatchPointer(0, 0, 0, 233, 300, 0,0,0,0,0,0,0)
.....

```

Figure 6.2: Excerpt of generated Monkey events

A list of 210 monkey events has been generated using the monkey event generator python script referred in Subsection 6.1.2. This list, composed of both touch events and key strokes, has then been used for the test hereby described. An excerpt of the script can be found in 6.2.

Test results are collected and processed in the same way as described in Subsection 6.3.1.

6.4 Test results

In this section test results will be presented, trying to propose a comparison of the generated system images as described in Subsection 6.3.1 and Subsection 6.3.2.

6.4.1 Comparison of DroidBox and instrumented GingerBread

A first test with the vanilla system image for Android 2.3 GingerBread directly from the Android SDK has been performed. The system image has been instrumented using the list shown in Algorithm 6.1.

A test with no stimulation has been executed on both the instrumented system image and DROIDBOX. The results have been collected and are shown in Figure 6.3. It is easy to notice that APIs related to network activity have a higher variance (as shown by the black bar which represents the standard deviation), this is generally due to advertisement libraries included in the applications that easily change their behaviour from an execution to another.

The same test has been repeated with input stimulation, as described in Subsection 6.3.2, the results are shown in Figure 6.4. Differently from the

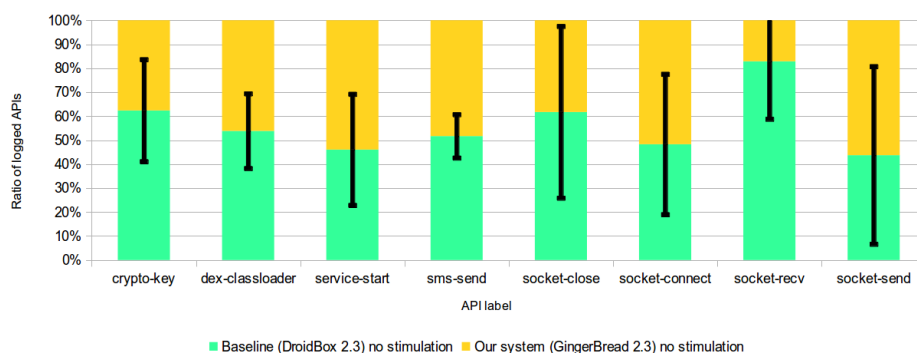


Figure 6.3: Comparison of DroidBox and GingerBread without stimulation

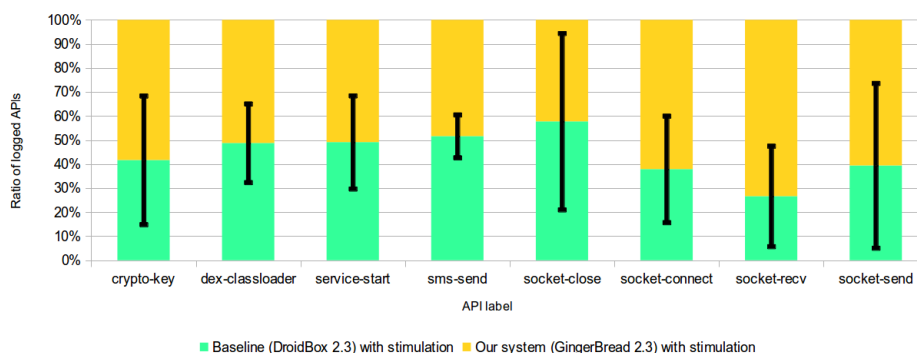


Figure 6.4: Comparison of DroidBox and GingerBread with stimulation

previous test GingerBread collected slightly more API invocations compared to DROIDBOX.

6.4.2 Comparison of DroidBox and GingerBread with GSF

The tests with and without stimulation have been run on the instrumented Android 2.3 GingerBread system image with the Google Services Framework installed. Test results comparing the system with DROIDBOX are shown in Figure 6.5 and Figure 6.6, corresponding to without and with stimulation respectively. It is possible to notice a growth, especially with respect to the `service-start` label, of the ratio of logged API by our system. This fact is justified by the presence of the Google Service Framework which offers several services that may be available at runtime. The two charts with and without stimulation are very similar, this may be a sign that the stimulation itself is not influencing API invocations as much as the presence of the Google Service

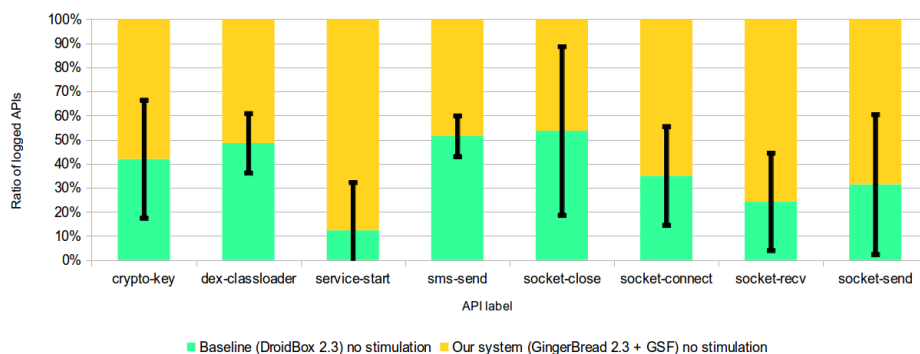


Figure 6.5: Comparison of DroidBox and GingerBread GSF without stimulation

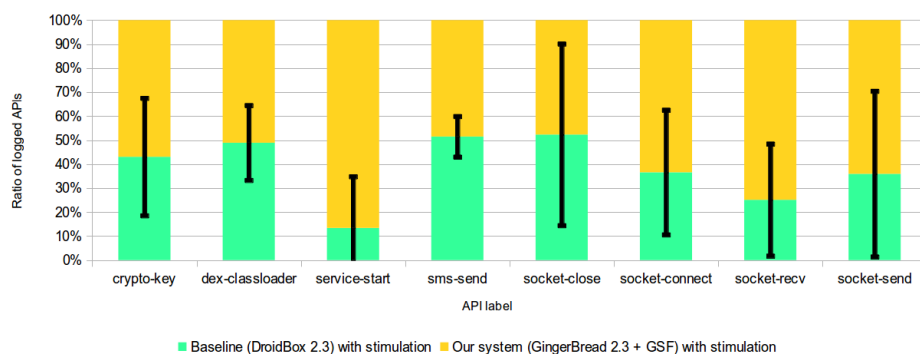


Figure 6.6: Comparison of DroidBox and GingerBread GSF with stimulation

Framework.

6.4.3 Comparison of DroidBox and ICS

The Android 4.0 Ice Cream Sandwich system image from the Android SDK has been instrumented using the API list shown in Figure 6.2. In this case different APIs have been instrumented, this is due to the fact that some of the APIs instrumented by DROIDBOX are no longer available in Android 4.0 Ice Cream Sandwich. An analysis of the class and call graph in the target system has been performed using the tool developed and described in Subsection 5.4.5. As a result, most of the APIs previously belonging to `Lorg/apache/harmony/luni/platform/OSNetworkSystem`; are now offered by the `Llibcore/io/IOBridge`; class.

Similarly to the previous section both tests with and without stimulation

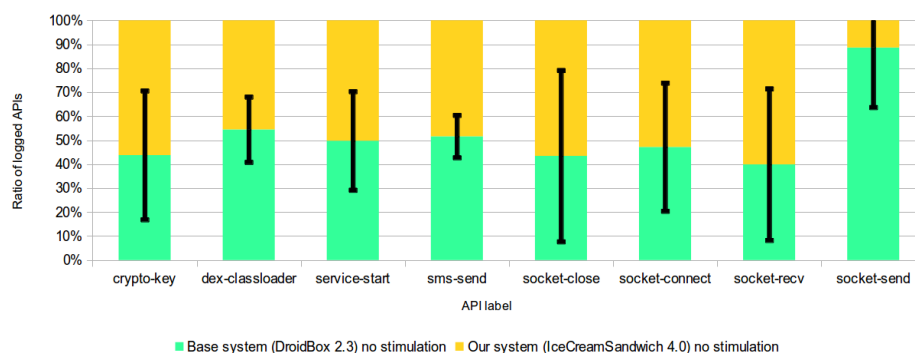


Figure 6.7: Comparison of DroidBox and ICS without stimulation

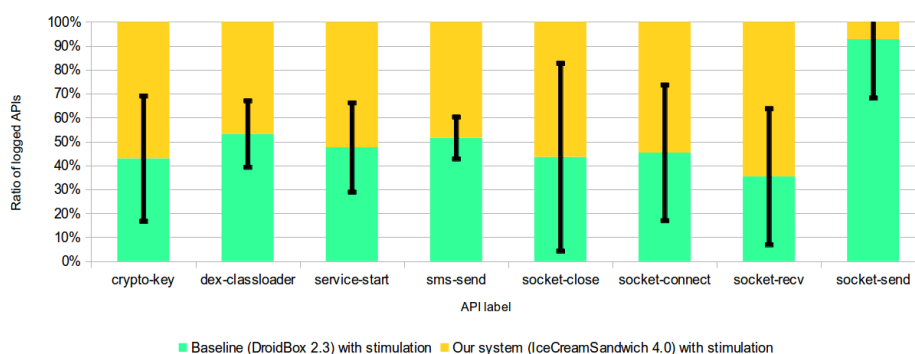


Figure 6.8: Comparison of DroidBox and ICS with stimulation

have been performed on this system and the results have been compared to the DROIDBOX ones, as shown in Figure 6.7 and Figure 6.8. The charts show that the two systems are balanced in terms of amount of logged API, apart from the `socket-send` label. This may be due to the different instrumented APIs, in fact the API instrumented in IceCreamSandwich 4.0 seems to not be equivalent to the one instrumented by DROIDBOX.

6.4.4 Impact of stimulation on test results

An interesting comparison is obtained estimating how the input stimulation affected the test results. This can show how effective the stimulation is with respect to the sample applications.

A first comparison was made with DROIDBOX, shown in Figure 6.9, in this case the stimulation mainly affected only network operations. A simple explanation for this phenomena may be that new content is retrieved from

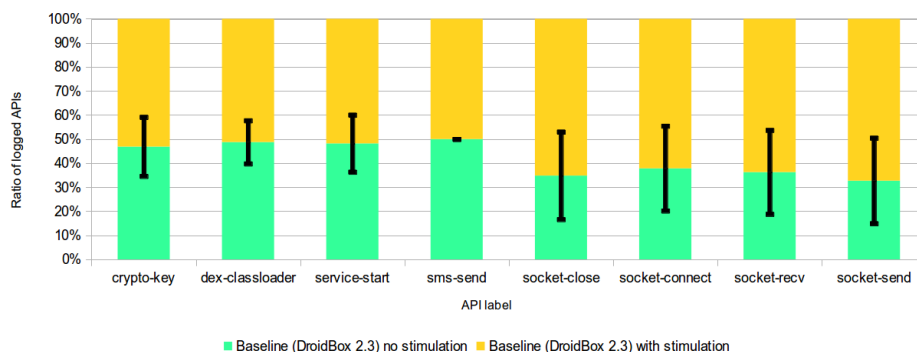


Figure 6.9: Comparison of DroidBox with and without stimulation

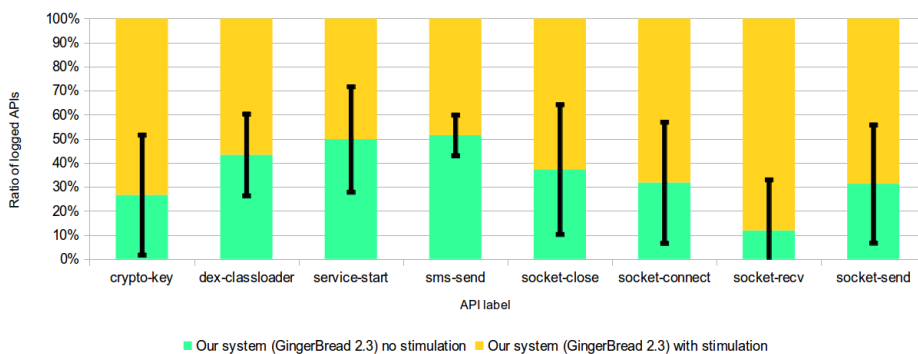


Figure 6.10: Comparison of GingerBread with and without stimulation

the web when touching UI elements, but this generally depends on the single application.

A second comparison was performed with the instrumented Android 2.3 GingerBread system: its test results with and without stimulation are shown in Figure 6.11. Similarly to DROIDBOX, network operations are increased, another rise is present in cryptography usage, this may be due to network communications over SSL such as HTTPS.

A third comparison was made using the GingerBread system image enriched with the Google Services Framework. Test results are shown in Figure 6.11, the trend shown is very similar to the one obtained with DROIDBOX.

A global view comparing all the systems is shown in Figure 6.13 and Figure 6.12 with and without stimulation respectively.

In both charts it is possible to notice how the systems behave similarly in APIs labelled as `crypto-key`, `dex-classloader` and `sms-send`. There is instead a

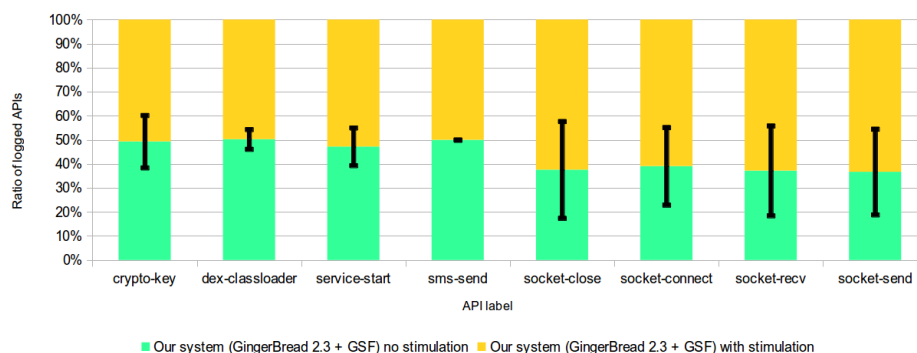


Figure 6.11: Comparison of GingerBread GSF with and without stimulation

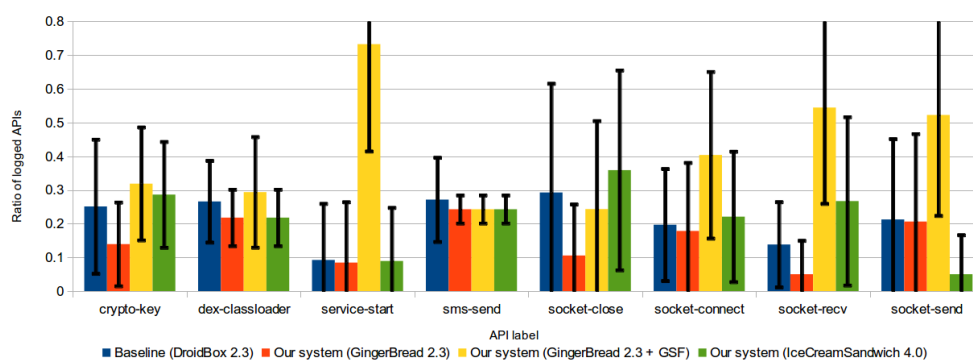


Figure 6.12: Comparison of the systems with no stimulation

prevalence of the GingerBread system with the Google Service Framework on the other APIs, which is due to invocations originating from services that comes with the framework.

6.4.5 Impact of the Google Services Framework

Here we will show how results differ when executing the same applications with or without the Google Services Framework installed.

This framework is used in Android devices when interacting with Google services such as Google Maps, Youtube or the Google Play Store.

Although real devices usually include these services, in the Android emulator they are not available. An attempt to create a more realistic environment has been made realising a system image that includes the Google Services Framework.

In Figure 6.14 are shown test results obtained with no input stimulation

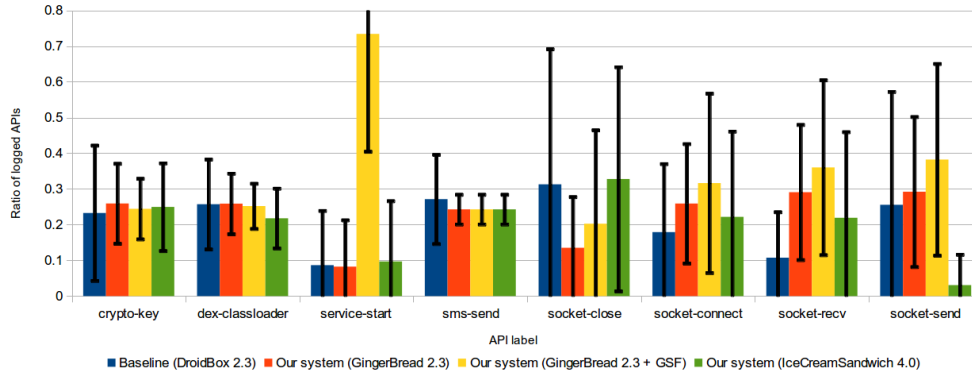


Figure 6.13: Comparison of the systems with input stimulation

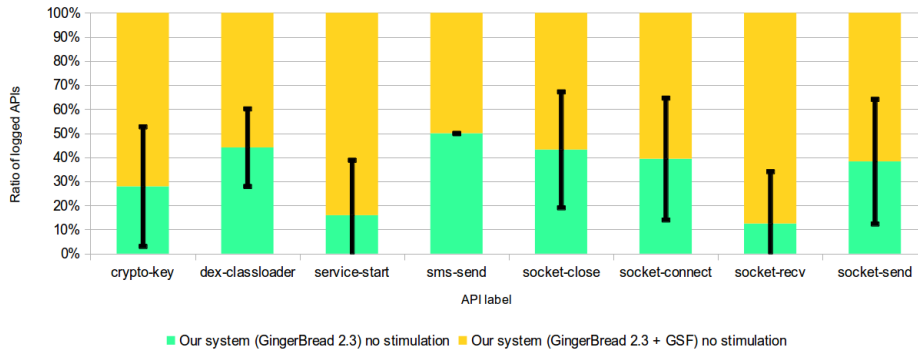


Figure 6.14: Impact of the Google Services Framework with no stimulation

that compare the instrumented Android 2.3 GingerBread without Google Services Framework with the one enriched with it.

The same procedure has been repeated with input stimulation, the results are shown in Figure 6.15.

In both charts a strong growth in services started is noticeable, this is probably due to applications invoking services belonging to the Google Services Framework.

6.4.6 Comparison of API usage in malicious and benign applications

As a last comparison test, the application samples have been partitioned in malicious and benign applications, a first test has been performed between DROIDBOX and the instrumented Android 2.3 GingerBread system image only

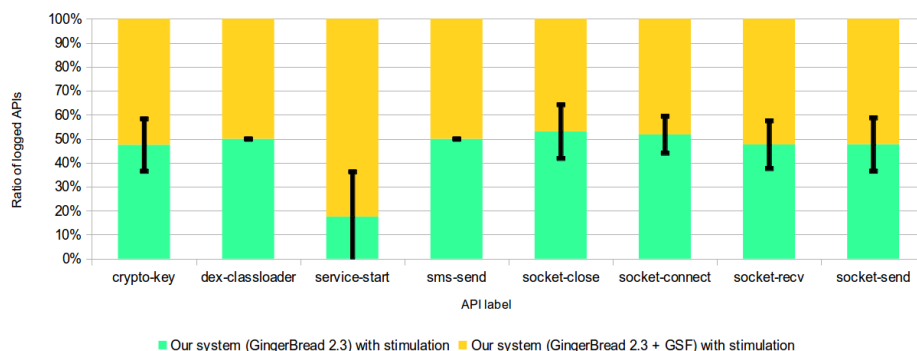


Figure 6.15: Impact of the Google Services Framework with input stimulation

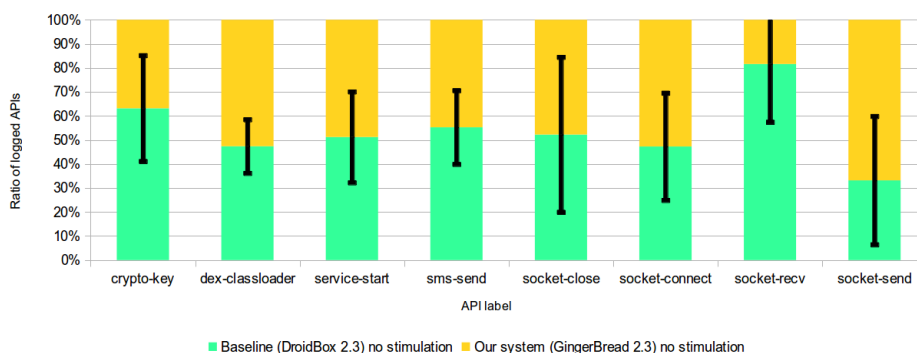


Figure 6.16: Comparison of DroidBox and GingerBread on malware samples

using malware application, and a second test using only benign applications.

The results of the first test, with malware applications, are shown in Figure 6.16, while results from the same test executed on benign samples is shown in 6.17. The comparison of the two systems does not differ much even if the two sample dataset are completely different, this means that the performance of the two systems in terms of logged APIs is not significantly affected by the type of application under test.

6.5 Discussion

In the previous section we presented the results obtained from the test that have been performed to evaluate our system. The evaluation considered four different system images, one of which is DROIDBOX. DROIDBOX is assumed as baseline system to compare the other system images, which are obtained

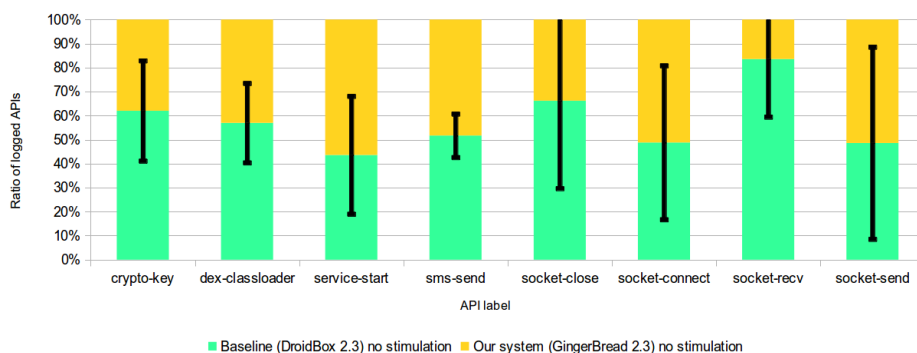


Figure 6.17: Comparison of DroidBox and GingerBread on benign samples

from the instrumentation of system images based on the stock Android version that comes with the Android SDK.

The main goal of the tests was to show how efficient our instrumented systems are in logging API calls while analysing a generic application. Since it is not possible to obtain an absolute measure, we run the tests on multiple systems and then compared each of the systems generated with our approach to current existing solutions to obtain an indicative measure.

The results showed that our system globally behave similarly to DROID-BOX, with some differences mainly on API related to network traffic. This is normally due to the fact that network data may vary from test to test since it is retrieved from external locations.

Other comparison are aimed to show differences from tests run on the same system with or without input stimulation described in Subsection 6.3.2. Results of these comparison showed a growth in network operations, probably due to network requests performed on input events, such as pressing a button or images loaded when changing screen.

A final comparison is made between a base system and a system on which the Google Services Framework have been installed. This framework includes functions specific to Google products such as Google Maps, Google Plus or Youtube, which may enable additional functions on some applications. The results of the comparison, shown in Figure 6.14 and Figure 6.15, demonstrates that the presence of the Google Services Framework slightly enhances the usage of some APIs, with a peak on the APIs representing the start of Services.

6.6 Other experiments

Other experiments have been performed just as proof of concept, without running the full set of tests described in the previous sections. Specifically DROIDSAGE correctly instrumented:

Physical device An *Acer Iconia Tab A500* tablet running *CyanogenMod 10*¹⁰, which is based on Android 4.1.2 *JellyBean*, was correctly instrumented using our approach. The tablet has been used for several hours for activities such as web browsing or playing resource intensive applications such as videogames without experiencing crashes.

x86 architecture Our system was able to perform instrumentation of an *X86* version of Android. The instrumented device is a virtual machine running Android 4.1.1 *JellyBean*.

The virtual device was obtained from the *AndroVM*¹¹ project, and the version that has been instrumented was the phone with Google Apps and other extras¹². It has to be noticed that no ODEX file was present in this system image, thus no conversion of ODEX files happened while generating the instrumented environment.

TaintDroid We successfully instrumented two system images obtained compiling Android sources with patches from the TAINTDROID project. The two images differ in the Android version, namely Android 2.3 *GingerBread* and Android 4.1 *JellyBean*. The generation of these system images proves that our approach can be considered orthogonal with respect to TAINTDROID, in fact the two systems coexist without interfering each other.

¹⁰<http://www.cyanogenmod.org/>

¹¹<http://androvm.org/blog/>

¹²http://androvm.org/Download/androVM_vboxx86p_4.1.1_r6.1-20130222-gapps-houdini-flash.ova

Chapter 7

Limitations and future works

This section presents the known limitations of the approach, as well as suggestions on possible extensions that have not been implemented for time constraints or because they were loosely related with the scope of this work.

Although the described approach has several interesting characteristics, some limitations apply. First of all the fact that it only targets and analyses Dalvik bytecode. While most of the APIs that interacts with the system force the application to perform invocations at bytecode level, there might exist cases in which the analyst is interested in monitoring native functions that are invoked by native libraries. In this scenario our approach can not be applied and therefore it can not fulfil the expected task.

Another minor limitation is the fact that only generic instrumentation hooks have been developed. In fact only logging hooks have been made available. A user interested in custom processing of the APIs has to develop his own hooks. This limitation has been accepted given that it is not possible to know in advance what a generic user requirements may be, thus customisation of the instrumentation hooks seems an acceptable limitation for the current work.

A further limit is the fact that this approach is strictly related to the Dalvik virtual machine and the current format of its bytecode. If newer version of both the virtual machine or the format of the bytecode will introduce significant changes, these would have to be taken into account and the existing code would have to be updated accordingly.

We acknowledge also the fact that the current approach transmits the execution trace of the hooked APIs through logging, this could be easily detected

by an application having the *android.permission.READ_LOGS*¹ permission set in its manifest file. Alternative approaches would require the ability to directly write files, transmit over the network or interacting with an application with such privileges.

Time constraints limited the testing of the instrumentation to tens of system images, an extensive test could be performed in order to find possible bugs in the instrumentation phase.

Another limit is the fact that even if the system can potentially run on multiple architectures with just minor changes, the conversion of ODEX files in their DEX equivalent currently works only on ARM architectures. This is because it would have required an additional effort in modifying the library used to parse and generate Dalvik bytecode.

As a last issue, our system may suffer code obfuscation techniques. More specifically our approach modifies the bytecode only in the places where instrumentation has to be performed, thus we transparently support obfuscation where no instrumentation is performed, but it may be a problem instrumenting a method whose name or hierarchy has been obfuscated.

¹https://developer.android.com/reference/android/Manifest.permission.html#READ_LOGS

Chapter 8

Conclusions

In this work we presented an automated system that generates dynamic analysis environment for the Android operating system. The resulting environments are suitable for blackbox analysis of unknown applications. The generation process of the environment is highly configurable, starting from a user supplied Android system image. The system image is then analysed and the Dalvik bytecode that is run by the Android operating system is instrumented. The instrumentation targets a customisable set of APIs whose bytecode is modified injecting code that will perform the tracing of invocations at runtime. Customisation can thus be applied both on the system images, provided as input to the system, and in terms of APIs that have to be instrumented. Furthermore, the resulting system image is resilient to signature verification. This happens by means of optimisation of the instrumented bytecode, in fact optimised bytecode present in ODEX files is not verified through signature verification, but just with the computation of an hash checksum.

Optimised bytecode is not only generated as output for the resulting environment, but also analysed if already present in the original system image. The analysis of optimised bytecode is a multi-step procedure described in Subsection 5.4.3 that aims to translate the optimised bytecode in its not optimised equivalent.

When the injected bytecode gets executed, instrumentation hooks are invoked. These are customisable methods that currently offer serialization to log of API name, parameters passed and stack trace at runtime invocation.

The system has been compared to the current *de facto* standard in Android applications blackbox dynamic analysis which is DROIDBOX (Subsection

2.2.2). Since our approach differs from the DROIDBOX one, the comparison has been performed only on the overlapping functions, namely logging of invoked APIs without taint tracking.

Further tests have been performed on different Android platforms in order to show how the underlying environment may impact on application behaviour.

Finally, we plan to release DROIDSAGE as an open source project. This choice aims to attract the scientific community, in which other projects may benefit by taking advantage of our work.

Appendix A

List of Dalvik VM instructions

The Dalvik virtual machine currently supports 246 different opcodes, each of them corresponding to a specific instruction. Every opcode belongs to a format, identified by a sequence of letters and numbers. The format identifier combines at least two numbers and a letter, the first number corresponds to the number of words (two bytes) which represent the instruction, the second one represents the number of parameters the instruction receives. A few instructions do not have a fixed number of registers as parameters, a range is used instead, in this case the letter “r” is used instead of the number of parameters. After these two numbers, a sequence of letters describes the parameter types:

b	immediate signed byte
c	constant pool index
f	interface constants
h	immediate signed hat (topmost 32 bits of a 64 bit value)
i	immediate signed int or float
l	immediate signed long or double
m	method constants
n	immediate signed nibble (4 bits)

s	immediate signed short value
t	branch target
x	no additional data

The most common instructions refer to registers only using nibbles, this fact limits the addressing of registers to only 4-bits.

A 4 bit addressing limits the register usage to only the first 16 registers, requiring to copy values from higher register numbers in order for them to be used. The same issue, with just different numbers, would apply when using an 8-bit addressing and more than 256 registers, it is just not common since methods normally do not use more than 20 registers.

Here follows a list of the opcodes supported by the Dalvik virtual machine.

Table A.1: Dalvik instruction set

00	10x	nop
01	12x	move vA, vB
02	22x	move/from16 vAA, vBBBB
03	32x	move/16 vAAAA, vBBBB
04	12x	move-wide vA, vB
08	22x	move-object/from16 vAA, vBBBB
09	32x	move-object/16 vAAAA, vBBBB
0a	11x	move-result vAA
0b	11x	move-result-wide vAA
0c	11x	move-result-object vAA
0d	11x	move-exception vAA
0e	10x	return-void
0f	11x	return vAA
10	11x	return-wide vAA
11	11x	return-object vAA
12	11n	const/4 vA, #+B
13	21s	const/16 vAA, #+BBBB
14	31i	const vAA, #+BBBBBBBB

Opcode	Format	Name
05	22x	move-wide/from16 vAA, vBBBB
06	32x	move-wide/16 vAAAA, vBBBB
07	12x	move-object vA, vB
15	21h	const/high16 vAA, #+BBBB0000
16	21s	const-wide/16 vAA, #+BBBB
17	31i	const-wide/32 vAA, #+BBBBBBBB
18	51l	const-wide vAA, #+BBBBBBBBBBBBBBBB
19	21h	const-wide/high16 vAA, #+BBBB000000000000
1a	21c	const-string vAA, string@BBBB
1b	31c	const-string/jumbo vAA, string@BBBBBBBB
1c	21c	const-class vAA, type@BBBB
1d	11x	monitor-enter vAA
1e	11x	monitor-exit vAA
1f	21c	check-cast vAA, type@BBBB
20	22c	instance-of vA, vB, type@CCCC
21	12x	array-length vA, vB
22	21c	new-instance vAA, type@BBBB
23	22c	new-array vA, vB, type@CCCC
24	35c	filled-new-array {vD, vE, vF, vG, vA}, type@CCCC
25	3rc	filled-new-array/range {vCCCC .. vNNNN}, type@BBBB
26	31t	fill-array-data vAA, +BBBBBBBB
27	11x	throw vAA
28	10t	goto +AA
29	20t	goto/16 +AAAA
2a	30t	goto/32 +AAAAAAAA
2b	31t	packed-switch vAA, +BBBBBBBB
2c	31t	sparse-switch vAA, +BBBBBBBB
2d	23x	cmpl-float
2e	23x	cmpg-float
2f	23x	cmpl-double
30	23x	cmpg-double
31	23x	cmp-long
32	22t	if-eq
33	22t	if-ne

Opcode	Format	Name
05	22x	move-wide/from16 vAA, vBBBB
06	32x	move-wide/16 vAAAA, vBBBB
07	12x	move-object vA, vB
34	22t	if-lt
35	22t	if-ge
36	22t	if-gt
37	22t	if-le
38	21t	if-eqz
39	21t	if-nez
3a	21t	if-ltz
3b	21t	if-gez
3c	21t	if-gtz
3d	21t	if-lez
44	23x	aget
45	23x	aget-wide
46	23x	aget-object
47	23x	aget-boolean
48	23x	aget-byte
49	23x	aget-char
4a	23x	aget-short
4b	23x	aput
4c	23x	aput-wide
4d	23x	aput-object
4e	23x	aput-boolean
4f	23x	aput-byte
50	23x	aput-char
51	23x	aput-short
52	22c	iget
53	22c	iget-wide
54	22c	iget-object
55	22c	iget-boolean
56	22c	iget-byte
57	22c	iget-char
58	22c	iget-short

Opcode	Format	Name
05	22x	move-wide/from16 vAA, vBBBB
06	32x	move-wide/16 vAAAA, vBBBB
07	12x	move-object vA, vB
59	22c	iput
5a	22c	iput-wide
5b	22c	iput-object
5c	22c	iput-boolean
5d	22c	iput-byte
5e	22c	iput-char
5f	22c	iput-short
60	21c	sget
61	21c	sget-wide
62	21c	sget-object
63	21c	sget-boolean
64	21c	sget-byte
65	21c	sget-char
66	21c	sget-short
67	21c	sput
68	21c	sput-wide
69	21c	sput-object
6a	21c	sput-boolean
6b	21c	sput-byte
6c	21c	sput-char
6d	21c	sput-short
6e	35c	invoke-virtual
6f	35c	invoke-super
70	35c	invoke-direct
71	35c	invoke-static
72	35c	invoke-interface
74	3rc	invoke-virtual/range
75	3rc	invoke-super/range
76	3rc	invoke-direct/range
77	3rc	invoke-static/range
78	3rc	invoke-interface/range

Opcode	Format	Name
05	22x	move-wide/from16 vAA, vBBBB
06	32x	move-wide/16 vAAAA, vBBBB
07	12x	move-object vA, vB
7b	12x	neg-int
7c	12x	not-int
7d	12x	neg-long
7e	12x	not-long
7f	12x	neg-float
80	12x	neg-double
81	12x	int-to-long
82	12x	int-to-float
83	12x	int-to-double
84	12x	long-to-int
85	12x	long-to-float
86	12x	long-to-double
87	12x	float-to-int
88	12x	float-to-long
89	12x	float-to-double
8a	12x	double-to-int
8b	12x	double-to-long
8c	12x	double-to-float
8d	12x	int-to-byte
8e	12x	int-to-char
8f	12x	int-to-short
90	23x	add-int
91	23x	sub-int
92	23x	mul-int
93	23x	div-int
94	23x	rem-int
95	23x	and-int
96	23x	or-int
97	23x	xor-int
98	23x	shl-int
99	23x	shr-int

Opcode	Format	Name
05	22x	move-wide/from16 vAA, vBBBB
06	32x	move-wide/16 vAAAA, vBBBB
07	12x	move-object vA, vB
9a	23x	ushr-int
9b	23x	add-long
9c	23x	sub-long
9d	23x	mul-long
9e	23x	div-long
9f	23x	rem-long
a0	23x	and-long
a1	23x	or-long
a2	23x	xor-long
a3	23x	shl-long
a4	23x	shr-long
a5	23x	ushr-long
a6	23x	add-float
a7	23x	sub-float
a8	23x	mul-float
a9	23x	div-float
aa	23x	rem-float
ab	23x	add-double
ac	23x	sub-double
ad	23x	mul-double
ae	23x	div-double
af	23x	rem-double
b0	12x	add-int/2addr
b1	12x	sub-int/2addr
b2	12x	mul-int/2addr
b3	12x	div-int/2addr
b4	12x	rem-int/2addr
b5	12x	and-int/2addr
b6	12x	or-int/2addr
b7	12x	xor-int/2addr
b8	12x	shl-int/2addr

APPENDIX A. LIST OF DALVIK VM INSTRUCTIONS

Opcode	Format	Name
05	22x	move-wide/from16 vAA, vBBBB
06	32x	move-wide/16 vAAAA, vBBBB
07	12x	move-object vA, vB
b9	12x	shr-int/2addr
ba	12x	ushr-int/2addr
bb	12x	add-long/2addr
bc	12x	sub-long/2addr
bd	12x	mul-long/2addr
be	12x	div-long/2addr
bf	12x	rem-long/2addr
c0	12x	and-long/2addr
c1	12x	or-long/2addr
c2	12x	xor-long/2addr
c3	12x	shl-long/2addr
c4	12x	shr-long/2addr
c5	12x	ushr-long/2addr
c6	12x	add-float/2addr
c7	12x	sub-float/2addr
c8	12x	mul-float/2addr
c9	12x	div-float/2addr
ca	12x	rem-float/2addr
cb	12x	add-double/2addr
cc	12x	sub-double/2addr
cd	12x	mul-double/2addr
ce	12x	div-double/2addr
cf	12x	rem-double/2addr
d0	22s	add-int/lit16
d1	22s	rsub-int (reverse subtract)
d2	22s	mul-int/lit16
d3	22s	div-int/lit16
d4	22s	rem-int/lit16
d5	22s	and-int/lit16
d6	22s	or-int/lit16
d7	22s	xor-int/lit16

Opcode	Format	Name
05	22x	move-wide/from16 vAA, vBBBB
06	32x	move-wide/16 vAAAA, vBBBB
07	12x	move-object vA, vB
d8	22b	add-int/lit8
d9	22b	rsub-int/lit8
da	22b	mul-int/lit8
db	22b	div-int/lit8
dc	22b	rem-int/lit8
dd	22b	and-int/lit8
de	22b	or-int/lit8
df	22b	xor-int/lit8
e0	22b	shl-int/lit8
e1	22b	shr-int/lit8
e2	22b	ushr-int/lit8
e3	22c	+iget-volatile
e4	22c	+iput-volatile
e5	21c	+sget-volatile
e6	21c	+sput-volatile
e7	22c	+iget-object-volatile
e8	22c	+iget-wide-volatile
e9	22c	+iput-wide-volatile
ea	21c	+sget-wide-volatile
eb	21c	+sput-wide-volatile
ec	00x	^breakpoint
ed	20bc	^throw-verification-error
ee	35mi	+execute-inline
ef	3rmi	+execute-inline/range
f0	35c	+invoke-direct-empty
f1	10x	+return-void-barrier
f2	22cs	+iget-quick
f3	22cs	+iget-wide-quick
f4	22cs	+iget-object-quick
f5	22cs	+iput-quick
f6	22cs	+iput-wide-quick

Opcode	Format	Name
05	22x	move-wide/from16 vAA, vBBBB
06	32x	move-wide/16 vAAAA, vBBBB
07	12x	move-object vA, vB
f7	22cs	+iput-object-quick
f8	35ms	+invoke-virtual-quick
f9	3rms	+invoke-virtual-quick/range
fa	35ms	+invoke-super-quick
fb	3rms	+invoke-super-quick/range
fc	22c	+iput-object-volatile
fd	21c	+sget-object-volatile
fe	21c	+sput-object-volatile

Bibliography

- [Fortune Tech(2011)] Fortune Tech. Industry first: Smartphones pass PCs in sales[M],[S.l.]: [s.n.] , 2011. <http://tech.fortune.cnn.com/2011/02/07/idc-smartphone-shipment-numbers-passed-pc-in-q4-2010/>.
- [US Computer Emergency Readiness Team(2011)] US Computer Emergency Readiness Team. Cyber Threats to Mobile Phones[R],[S.l.]: [s.n.] , 2011. https://www.us-cert.gov/sites/default/files/publications/cyber_threats-to_mobile_phones.pdf.
- [Egele et al.(2008)] Egele M, Scholte T, Kirda E, et al. A survey on automated dynamic malware-analysis techniques and tools[J]. ACM Comput. Surv., 2008, 44(2):6:1–6:42. http://iseclab.org/papers/malware_survey.pdf.
- [Bornstein(2008)] Bornstein D. Dalvik vm internals[C]. In: Google I/O Developer Conference. 2008. 23:17–30. http://fiona.dmcs.pl/podyplomowe_smtm/smob3/Presentation-Of-Dalvik-VM-Internals.pdf.
- [Felt et al.(2011)] Felt A P, Chin E, Hanna S, et al. Android permissions demystified[C]. In: Proceedings of the 18th ACM conference on Computer and communications security. 2011. 627–638. <http://www.cs.berkeley.edu/~daw/papers/androidperm-ccs11.pdf>.
- [Enck et al.(2010)] Enck W, Gilbert P, Chun B G, et al. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones.[C]. In: OSDI. 2010. 10:255–270. http://static.usenix.org/events/osdi10/tech/full_papers/Enck.pdf.
- [Cavallaro et al.(2008)] Cavallaro L, Saxena P, Sekar R. On the limits of information flow techniques for malware analysis and containment[M]//.

- In: Detection of Intrusions and Malware, and Vulnerability Assessment.[S.l.]: Springer, 2008. 143–163. <http://seclab.cs.sunysb.edu/seclab1/pubs/antitaint.pdf>.
- [Bayer et al.(2009)] Bayer U, Habibi I, Balzarotti D, et al. A view on current malware behaviors[C]. In: USENIX workshop on large-scale exploits and emergent threats (LEET). 2009. https://www.usenix.org/legacy/event/leet09/tech/full_papers/bayer/bayer_html/.
- [Yan and Yin(2012)] Yan L K, Yin H. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis[C]. In: Proceedings of the 21st USENIX Security Symposium. 2012. <https://www.usenix.org/system/files/conference/usenixsecurity12/sec12-final107.pdf>.
- [Reina et al.(2013)] Reina A, Fattori A, Cavallaro L. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors[J]. EUROSEC, Prague, Czech Republic, 2013. <http://security.dico.unimi.it/~joystick/pubs/eurosec13.pdf>.
- [Rastogi et al.(2013)] Rastogi V, Chen Y, Enck W. AppsPlayground: automatic security analysis of smartphone applications[C]. In: Proceedings of the third ACM conference on Data and application security and privacy. 2013. 209–220. <http://www.enck.org/pubs/codaspy13.pdf>.
- [Ramsdell(2010)] Ramsdell T. S/MIME version 3.2 message specification[J]. 2010. <http://xml2rfc.tools.ietf.org/pdf/rfc5751.pdf>.
- [Barrera et al.(2012)] Barrera D, Clark J, McCarney D, et al. Understanding and Improving App Installation Security Mechanisms Through Empirical Analysis of Android[C]. In: Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices. 2012. SPSM '12. <http://doi.acm.org/10.1145/2381934.2381949>.
- [Sima(2000)] Sima D. The design space of register renaming techniques[J]. Micro, IEEE, 2000, 20(5):70–83. <http://classes.soe.ucsc.edu/cmpe202/Fall10/papers/rat.pdf>.
- [Theiling(2000)] Theiling H. Extracting safe and precise control flow from binaries[C]. In: Real-Time Computing Systems and Applications,

2000. Proceedings. Seventh International Conference on. 2000. 23–30. <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.25.279&rep=rep1&type=pdf>.
- [Cytron et al.(1991)] Cytron R, Ferrante J, Rosen B K, et al. Efficiently computing static single assignment form and the control dependence graph[J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 1991, 13(4):451–490. <https://www.cs.cmu.edu/afs/cs/academic/class/15745-s07/www/papers/cytron-efficientSSA.pdf>.
- [Zhou and Jiang(2012)] Zhou Y, Jiang X. Dissecting android malware: Characterization and evolution[C]. In: Security and Privacy (SP), 2012 IEEE Symposium on. 2012. 95–109. <http://web1.cs.columbia.edu/~nieh/teaching/e6998/papers/OAKLAND12.pdf>.