Mestrado em Engenharia Informática
Dissertação
Relatório Final

Digging deeper: Hardware-supported dynamic analysis of Linux Programs

Rodrigo Oliveira Santos
rosantos@student.dei.uc.pt

Orientador:
Prof. Federico Maggi

Co-orientador:
Prof. Jorge Granjal

02 de Setembro de 2015

FCTUC DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Examiners
Professor Edmundo Monteiro
Professor Marco Vieira

I would like to dedicate this thesis to my parents...

Acknowledgements

Firstly, I would like to express my gratitude to both my advisors, Prof. Federico Maggi from Politecnico di Milano Prof. Jorge Granjal from University of Coimbra for their continuous guidance, availability, support and motivation throughout this challenging project. A special thanks to Prof. Federico Maggi for taking the risk of advising a master thesis of a former Erasmus student.

Secondly, I thank my friends and colleagues which shared the experience of studying in Coimbra and made me go through one of the best experiences of my life. This journey full of sleepless nights would not be the same without the funny and bohemian moments which we shared.

Next, I would like to thanks to my girlfriend for her support and encouragement throughout this major and for understanding when I lacked availability. In addition, I would like to thank her guidance and patience in all my challenges and for always being there for me.

Last, but not least, I would like to thank my parents for all the support and comprehension in the hardest times. They always stood for me and supported me during this adventure, even on tough decisions like being without them for six months. I would like to thank them for allowing me to make the most of this experience.

Abstract

Malware has been evolving and now, besides computers, is targeting smartphones and tablets. This evolution has occurred because mobile devices contain sensitive data and are becoming ubiquitous, which is a substantial threat for users. Dynamic program-analysis techniques are used to observe the actions taken by a malicious application, to recognize signs of malicious sequences. Typically, dynamic tracing leverages a virtualized or emulated environment to observe the execution while being invisible to the malware. However, there are several techniques that malware can exploit to figure out that it is running under a virtual environment, and show no malicious behavior, such as to appear benign to the observer.
We propose to eradicate the aforementioned problem, by creating an hardware-based solution for runtime events reconstruction. The rationale is that, being running on real hardware, the existing environment-fingerprinting techniques are less likely to succeed. Our idea is to leverage the availability of low-cost development boards and smartphones, equipped with JTAG debugging interfaces, that run Android. Building upon the introspection data that can be extracted via the JTAG interfaces (e.g., CPU instructions, hardware state, power consumption), we will adapt existing techniques to reconstruct the runtime events (e.g., system calls) of a process from hardware-derived instruction traces, which is the first step for creating a more robust dynamic analysis tool.

Keywords: JTAG, Android, Dynamic Analysis, Debugging, Tracing, OpenOCD, Pandaboard ES, Flyswatter2, Linux

Resumo

O malware, ao longo dos anos, tem vindo a evoluir e, atualmente, alm de computadores, est a comear a alcanar dispositivos mveis como smarphones e tablets. Esta evoluo ocorre porque os dispositivos mveis contm dados sensveis e so cada vez mais ubquos, o que representa uma enorme ameaa para os utilizadores. Tcnicas de anlise dinmica so usadas para observar as aes tomadas por uma dada aplicao maliciosa e reconhecer as sequncias maliciosas nela inseridas. Tipicamente, estas tcnicas instrumentam ambientes virtualizados ou emulados para observar a execuo de uma aplicao sem que esta se aperceba que est a ser analisada. No entanto, existem algumas tcnicas que permitem que o malware se aperceba que est a executar num ambiente virtualizado, no executando assim, as suas sequncias maliciosas para ludibriar o utilizador.
Propomos, para erradicar o problema supracitado, a criao de uma soluo baseada em hardware de modo a reconstruir chamadas ao sistema em tempo de execuo. O raciocnio passa por assumir que ao executar a anlise num ambiente real as tcnicas existentes de reconhecimento de ambientes virtualizados tero menos hipteses de ser bem-sucedidas. O objetivo do presente trabalho foca-se em tirar partido de placas de custo reduzido equipadas com interfaces JTAG que suportem Android. Ao reconstruir eventos de tempo de execuo de um processo, como chamadas ao sistema, atravs de informao extrada a partir de interfaces JTAG, estamos a adaptar tcnicas existentes com dados recolhidos diretamente do hardware. Este deve ser o primeiro passo para a criao de uma ferramenta de anlise dinmica mais robusta.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Android OS is becoming more and more popular in a wide range of electronic devices (smartphones, tablets, embedded systems, etc). On the smartphone market, only in the third quarter of 2014, 283 million units were shipped, which represents 84% of the market share [IDC, 2014]. On top of that, there are over 1 million new Android devices activations worldwide [Android-central, 2015]. This popularity, its open model and the users sensitive data, positions Android OS on an high threat level, since it is very attractive for cyber-attackers. Moreover, the easiness of uploading malware to mobile marketplaces [W. Zhou and Ning, 2012] and the highly delayed patches from producers worsen the users security. Furthermore, according with mcafee [McAfee, 2014], the number of new malware, per quarter of an year, is increasing over time. More precisely, more than 700 000 new malware samples were found in the first quarter of 2014, which gives a total of, roughly, 4 000 000 malware samples at that time. And the trend is to increase even more. Nowadays malware is mainly profit-motivate, more precisely, 88% of the all samples, in the first quarter of 2014 [F-SecureLabs, 2014], is targeting users sensitive information, because it is highly valuable in the black market [Symantec, 2012]. Threats like sending SMS and dialing to premium numbers, billing of fake antivirus (ransomware), bypassing two-factor authentication, among others are now real in mobile devices. A recent study shows the monetization system, through mobile malware, on the underground economy of China [trend micro, 2014].

Under such scenario, it was mandatory for security experts to create tools to tackle the previous issues. Current tools, that detect hidden malicious activities, are based in two techniques: static analysis [DEXLabs, 2012] and dynamic analysis [InternationalSecureSystemsLab, 2012; K. Tam and Cavallaro, 2015; the honeynet project, 2012; Yan and Yin, 2012]. Google has created Bouncer [Google, 2012], which tries to detect malicious Applications, but it can be fingerprinted and evaded [Oberheide and Miller, 2012].

Research Gap However, cyber-atackers have techniques to evade current techniques. Static analysis might be evaded by obfuscating code [V. Rastogi and Jiang., 2013]. Additionally, it cannot record runtime information that may be useful. Dynamic analysis tools overcome the limitations of static analysis. These tools often rely on emulators or virtual machines and leverage the capabilities of VMI [Garfinkel and Rosenblum, 2003] to analyze attacks even on the kernel. Nevertheless, such environments can be fingerprinted, making it possible for the malware to hide its malicious activities [Oberheide and Miller, 2012; T. Petsas and Ioannidis, 2014; T. Vidas, 2014].

The project hereby presented intends to take the next step into dynamic analysis tools. We are developing an automated tool that thwarts fingerprinting existing techniques by recording informations directly from an ARM-based CPU, since it is the most widely adopted standard for smartphones [ARM®, 2015], and by introspecting the main memory. This will basically raise the bar to existing fingerprinting techniques employed by malware, since the environment is not virtualized or emulated. The recorded information will then be used to reconstruct the behavior of a given software.

The novelty of this approach lies in the reconstruction the runtime events, like system calls, by observing directly a real CPU and the main memory, with the aid of JTAG protocol and ARM's debug IP [1], instead of instrumenting an emulated CPU through virtual machine introspection techniques. Furthermore, it

[1]IP: Intelectual property

will prevent most, if not all, the existing fingerprinting techniques, so that it will be harder for malware to hinder analysis.

1.1 Objectives

This work aims to create a cost-effective automated solution which analyzes the behavior of a specific Linux process. This solution should receive a program to seamlessly install, execute and analyze a given program. The analysis should be saved to a file so that an expert can read and infer the behavior of the program.

Chapter 2

State of the Art and Motivation

This chapter supplies information on the state of the art of dynamic analysis, its benefits and its limitations. Additionally, it analyzes the used tools [K. Tam and Cavallaro, 2015; Yan and Yin, 2012] based on this technique, that can be applied on Android operating system, since they are the most interesting ones for this work scope.

The first section provides an overview of dynamic analysis itself and virtual machine introspection (VMI) [Garfinkel and Rosenblum, 2003]. Then it is divided into two subsections: available approaches [K. Tam and Cavallaro, 2015; Yan and Yin, 2012]; evasion techniques [T. Petsas and Ioannidis, 2014; T. Vidas, 2014]. The first subsection provides additional information about the tools used to employ dynamic analysis in Android OS. In the second subsection, the reader can have an insight of the known existing techniques employed to evade dynamic analysis-based existing tools.

2.1 Dynamic Analysis

Dynamic Analysis is a technique that infers the behavior of a given software, in an automated way, by analyzing its actions. Security tools leverage this analysis as a data source to classify software behaviors as malicious (malware) or benign. The way that the analysis is performed may vary on the behavior-based technique employed. The next topics describe two different tools [K. Tam and Cavallaro, 2015; Yan and Yin, 2012] that are examples of those variations, none of which is better than the other, so it really depends on what the security expert wants to analyze or do. However, in general, dynamic-based techniques monitor events that define the program's flow, infer its behaviors from these events and detect high-level malicious behaviors.

The major advantages of this technique are the resilience to code obfuscation, via encryption of the source code or packing [1] and the easiness of the analysis. Dynamic analysis tackles this because it inspects the software's behavior and not its source code. As for the downsides, dynamic analysis, in most of the cases, cannot analyze all the code, since it can only see what is being executed by a given software. Thus, the software (malicious or benign at the time) can have one or more behaviors that are not seen at execution time. Additionally, instrumentation can usually be detected, that is, it is possible for a malware to detect that it is being debugged or analyzed.

2.2 Virtual Machine Introspection overview

There is a spectrum of different dynamic-analysis techniques to use, depending on the level at which the observation of the events is performed (e.g. CPU, drivers, operating system, application). Most of the modern tools [InternationalSecureSystemsLab, 2012; K. Tam and Cavallaro, 2015; the honeynet project, 2012; Yan and Yin, 2012] employ out-of-the-box analysis, which performs dynamic analysis to analyze a given malware by inspecting sandbox environments (virtual machines or cpu emulators like QEMU [Bellard, 2005]) from the outside. Out-of-the-box analysis is well known as virtual machine introspection.

VMI, as a dynamic analysis technique, inherits its benefits and limitations. Furthermore, this kind of introspection leaves few, if none, artifacts that the malware can see. Thus, it is harder for the malware to fingerprint the presence of

[1]Packing refers to the application of different encryption methods on the source code

the observation put in place. However, it might be heavier to do such analysis, because it is creating another level of abstraction.

The following subsections present the two most important out-of-the-box dynamic analysis tools for Android OS: DroidScope and CopperDroid.

2.3 Available Approaches

2.3.1 Android background

Android OS (which is linux-based) creates a different process for each an application (App). On its side, the App is within a runtime virtual machine also known as Dalvik Virtual Machine (DVM). This virtual machine supplies a run-time environment for the Java components of the App. Furthermore, it is possible to execute native code through the Java Native Interface (JNI). The figure 2.1 depicts this abstraction level.

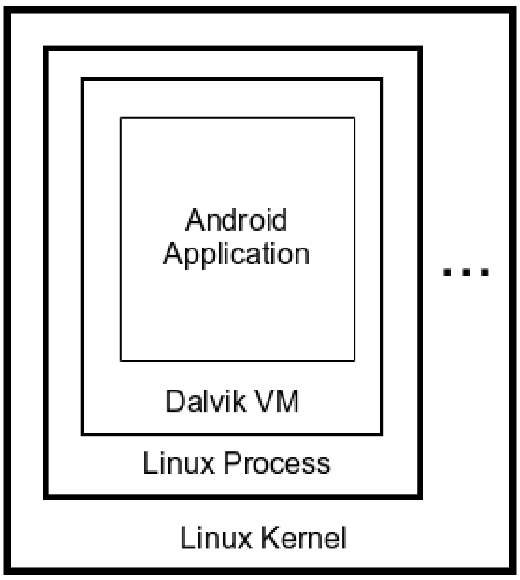


Figure 2.1: Android abstraction level

2.3.2 DroidScope

DroidScope is a platform that that performs out-of-the-box (VMI-based) Android malware analysis. Porting such analysis technique from desktop, even though that Android OS is Linux-based on the lower level, is no easy task. This platform relies on the Android's abstraction levels 2.1 and reconstructs both OS-level and Java-level semantics. While OS-level semantics are the behaviors of the malware process and its native components, Java-level semantics are the behaviors of Java components. DroidScope provides three tiered APIs that mirrors hardware, OS and Dalvik VM, and which enables the possibility of building tools on top of it. In order to demonstrate some of the capabilities of this platform, it supplies analysis tools to collect native and Dalvik instruction traces, log applications' interactions with Android OS and perform taint analysis to check information leakage.

The API tracer monitors the malwares activities at the API level to reason about how the malware interacts with the Android runtime environment. This tool monitors how the malwares Java components communicate with the Android Java framework, how the native components interact with the Linux system, and how Java components and native components communicate through the JNI interface. The DroidScope's architecture is depicted in the figure 2.2
DroidScope is built on top of QEMU CPU emulator [Bellard, 2005], which provides an VMI-based analysis. Thus, the changes are made in the emulator and not in the Android OS. Additionally, QEMU allowed the support of many devices as well as different architectures (e.g. ARM and x86), after some alterations (e.g. in the registers because they are different).

DroidScope reconstructs the OS-level view to analyze native components. To achieve this reconstruction DroidScope performs some basic instrumentation similar to the existing VMI-based techniques on x86 architecture [Garfinkel and Rosenblum, 2003]. QEMU uses dynamic translation with the aid of an intermediate representation named Tiny Code Generator (TCG). DroidScopes achieves analysis by inserting extra TCG instructions to retrieve additional information for further analysis. In order to infer a user-level behavior, like file, network accesses and interprocess communication, some system calls its return values are fetched as well. DroidScope leverages the capabilities of *task_struct*'s list (which contains every task active and it is depicted in the figure 2.3) to trace information about which processes and threads are active and which one of them is actually executing.

Lastly, since Dalvik VM, libraries and dex files are mapped in memory, DroidScope gets access to the memory map of a process using *mmap* pointer, which is accessible though *task_struct* as well. It is worth noting that it also keeps track of *sys_mmap2* system call to update memory map when it returns.

DroidScope reconstructs the Dalvik view (Dalvik instructions, current machine state, Java objects) using low-level semantics and the *mterp* interpreter. In order to access Dalvik instructions, this platform uses the *mterp* interpreter, that by using an offset-addressing method, maps Dalvik opcodes to machine code blocks. Each opcode has 64 bytes of memory to store the corresponding emulation code. Such a design makes the reverse conversion straightforward: if the program counter(*R15*) points to any of these code regions, then DVM is interpreting a byte-code instruction. DroidScope identifies the virtual address of rIBase (starting point of the code

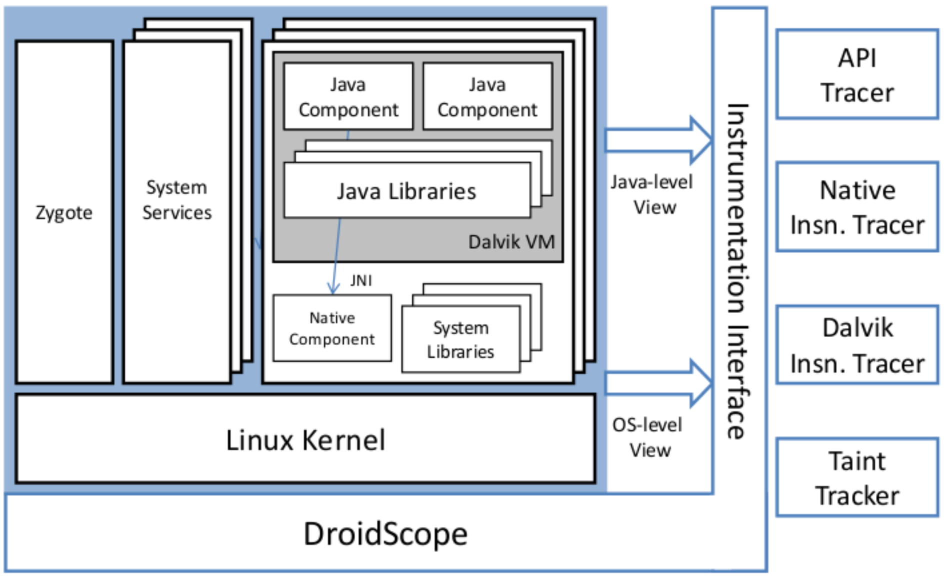


Figure 2.2: DroidScope Overview [Yan and Yin, 2012]

block) using information gathered from the OS-level view and then calculates the opcode using the following formula:

$$\frac{R15 - rIBase}{64}$$

However, it is also possible to translate Dalvik bytecode instructions with the Just-In-Time compilation (JIT), which increases the performance, because it caches the most used blocks, but makes it harder the instrumentation. It was against the requirements to disable JIT at build time, since it would require further changes to the virtual device. The adopted solution was to selectively disable JIT at runtime. DroidScope enables analysis plugins to select code regions for which they want to disable JIT. Every analyzed Dalvik block will cause a performance penalty, since it will not be cached for sure. As the analyzed code is not cached, it is possible for the *mterp* interpreter to emulate the code.

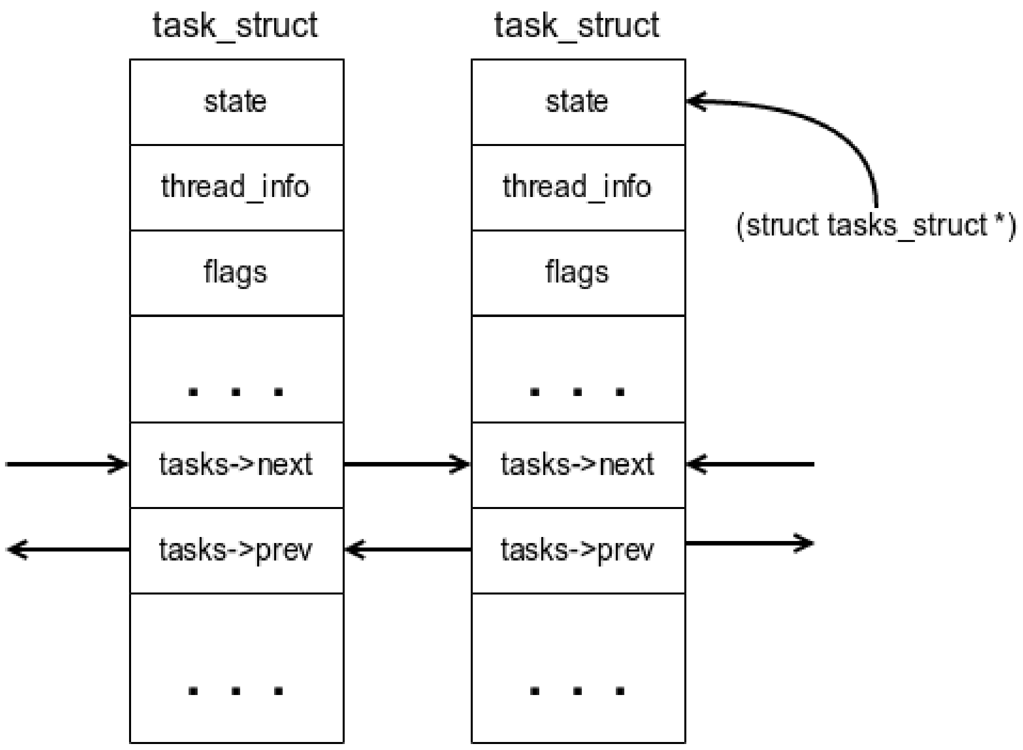


Figure 2.3: task_struct's list

Dalvik VM state is stored in the CPU registers. On ARM the used registers goes from *R4* to *R8* and contain the following information:

Table 2.1: DVM execution context

	Information
R4	Dalvik program counter
R5	Dalvik stack frame pointer
R6	*glue*
R7	First two bytes of the current Dalvik instruction
R8	base address of the *mterp* emulation code for the current DVM instruction

In the previous table, *glue* stands for *InterpState* data structure, which stores information like return value and thread information.

The reconstruction of Java Objects is done with the aid of two data structures: *ClassObject* and *Object*. The *ClassObject* data structure contains a class type and important information like the class name, where it is defined in a dex file, the size of the object, the methods, and the location of the member fields within the object instances. It is generic enough to describe the class types and its implicit class types.

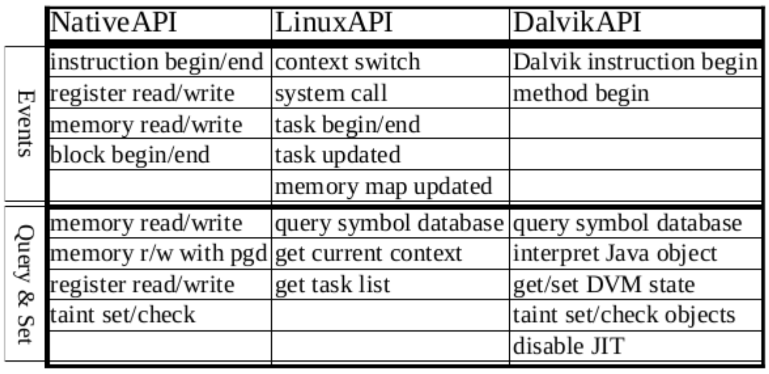
On the other hand, *Object* describes the contents of the instances created at runtime. Each *Object* has a pointer to the *ClassObject* that represents its type and stores further information about that instance's contents.

For the sake of human readability, DroidScope manages a symbol database with symbols like functions names, classes names, fields names. Furthermore it stores a database of offsets so that it is possible to access symbols even when the system has Address space layout randomization (ASLR). Finding a symbol at runtime requires two steps: identifying the containing module using the shadow memory map; calculating the offset to search the database. As for the Native libraries, symbols are gathered with *objdump*. To ensure the best symbol coverage, *dexdump* is employed when Dalvik or Java symbols cannot be retrieved dynamically, for

instance, in the case that the corresponding page of a dex file is not loaded in memory yet.
DroidScope offers basic hooking mechanisms by means of the APIs that supply instrumentation on three different levels: native, OS and Dalvik. The following figure presents its features:

2.3.3 CopperDroid

CopperDroid is a system call-centric tool built on top of QEMU [Bellard, 2005] that performs out-of-the-box dynamic analysis and reconstructs the behaviors of Android malware.
From the Android abstraction level figure 2.1, it seems that dynamic system-call malware analysis systems on Android OS cannot be built as they are in desktop environments. In fact, as a previous platform demonstrated [Yan and Yin, 2012], in order to build such systems there are two levels of semantic information that must be reconstructed: the high-level (application actions) and low-level (OS actions). Additionally, the previous tool demonstrated that traditional system call-centric dynamic analysis is not suited to analyze malware, since it lacks information on the Android-specific semantics.
From a different perspective, CopperDroid relies on Binder protocol to infer all the semantic information, needed (OS and Dalvik views) to reconstruct a malware's behavior. In a nutshell, binder protocol is an optimized synchronous inter-process communication (IPC) and remote procedure call (RPC) mechanism. On Android



	NativeAPI	LinuxAPI	DalvikAPI
Events	instruction begin/end	context switch	Dalvik instruction begin
	register read/write	system call	method begin
	memory read/write	task begin/end	
	block begin/end	task updated	
		memory map updated	
Query & Set	memory read/write	query symbol database	query symbol database
	memory r/w with pgd	get current context	interpret Java object
	register read/write	get task list	get/set DVM state
	taint set/check		taint set/check objects
			disable JIT

Figure 2.4: DroidScope APIs [Yan and Yin, 2012]

OS, every process has a binder thread so they can communicate with the its driver module. The communication processes as follows: For every transaction that a component A of an App wants to send to a component B (whether that component is located in the same App, in a different App or in the kernel), an *ioctl* system call is raised. The *ioctl* system call is then handled by the Binder kernel driver.

Every transaction sends a *Parcel*, which is a container for a message of marshalled [1] data and meta-data. The receiver process must *unmarshall* the data in order to get the *Parcel*'s contents.

Processes must understand the communication through binder protocol, that is, they should have a way to access an interface that allows applications to communicate with each other through services. Services are defined by Android Interface Definition Language (AIDL), that defines which methods can be invoked and what parameters' type they receive.

A naive analysis of system calls on Android OS does not provide enough semantic information to reconstruct the malware's behavior, however CopperDroid leverages the capabilities of Binder protocol to reconstruct the Android-specific semantics, since every request and data exchange go through Binder. Additionally, even when passing-by-reference the data itself is sent through IPC channels in a flattened Binder. Thus, CopperDroid enables the reconstruction of behaviors of

[1] Marshalling refers to the procedure for converting high-level Android-specific data structures into *parcels*

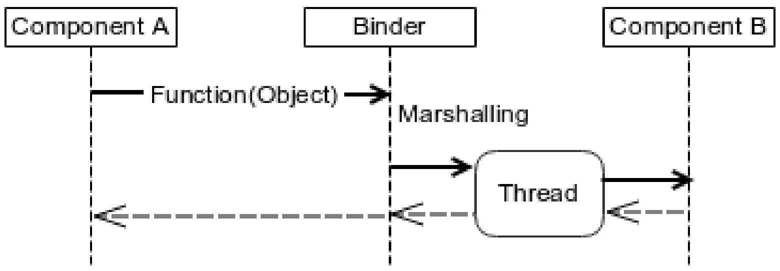


Figure 2.5: Binder Communication

Android Apps at multiple levels of abstraction from the observation of system calls.

CopperDroid is Android version independent, that is, it is a generic solution that does not need to keep in sync Dalvik and OS views. The best example of this abstraction is the successful analysis with Android Runtime (ART) instead of DVM without changing the instrumentation. Lastly, in order to enhance the analysis, a stimulation application [1] is used with stimulation techniques that are known to trigger malicious events.

[1]Monkeyrunner: http://developer.android.com/tools/help/monkeyrunner concepts.html

The following figure provides an overview of CopperDroid architecture: CopperDroid is composed by the following components: CopperDroid emulator, CopperDroid behavior reconstruction analysis, vanilla android emulator. The CopperDroid emulator is built on top of QEMU and it is instrumented to trap *ioctl* system calls (Binder transactions) for ARM-based and x86-based processors. CopperDroid parses the Binder transaction, extracting the interface token and the arguments. Afterwards, it sends the *marshalled arguments* and its types to the vanilla android emulator (unmarshalling oracle). This oracle unmarshalls the received arguments and sends them back for further analysis. It is worth noting, that this process is not very intrusive, in the way that it only needs to trap syscalls from the CopperDroid emulator. The syscalls are then sent to the out-of-the-box analysis.

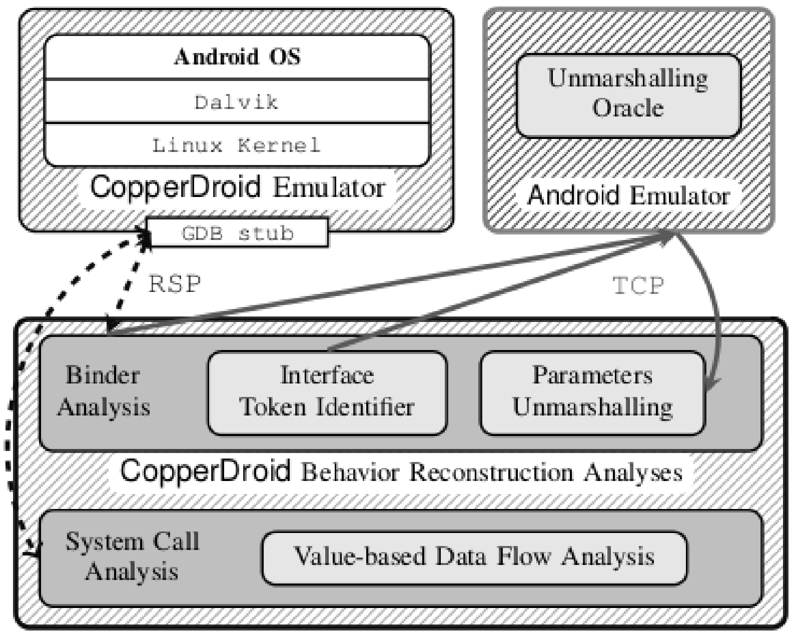


Figure 2.6: CopperDroid architecture [K. Tam and Cavallaro, 2015]

2.4 Evasion Techniques

Dynamic analysis techniques and, in particular, VMI-based techniques, as a complementary and important analysis, have become a standard for security experts. Tools like the ones described in the last section have emerged and corporations are using them to inspect malicious behaviors on Android Apps. This trend led malware writers ways to fight back this tools. On this line of reasoning they created *evasive malware*. This kind of malware has the ability to detect virtualized environments [R. Paleari and Bruschi, 2009] and allows the malware to hide its malicious behavior. There are two ways used to evade dynamic analysis: *red pill*; *blue pill*. The first one detects whether the Application it is running on a virtualized environment, or not. The second one is a virtualized *rootkit* that creates a thin hypervisor, so that the all OS is virtualized, which makes the malware impossible to with analysis tools because it can fool them [Rutkowska, 2006]. Currently, malware writers use *red pills* to detect the kind of environment where they are running on. Thus, they can hinder such environments by crashing themselves or obfuscate their behavior. The assumption is as follows: if the malware is running on a virtual machine it is probably being analyzed.

This subsection supplies the existing known techniques that are employed by Android malware writers in order to fingerprint the environment (*red pills*) and evade analysis. These techniques are employed to evade virtualized or emulated systems, by obfuscating their malicious behaviors.

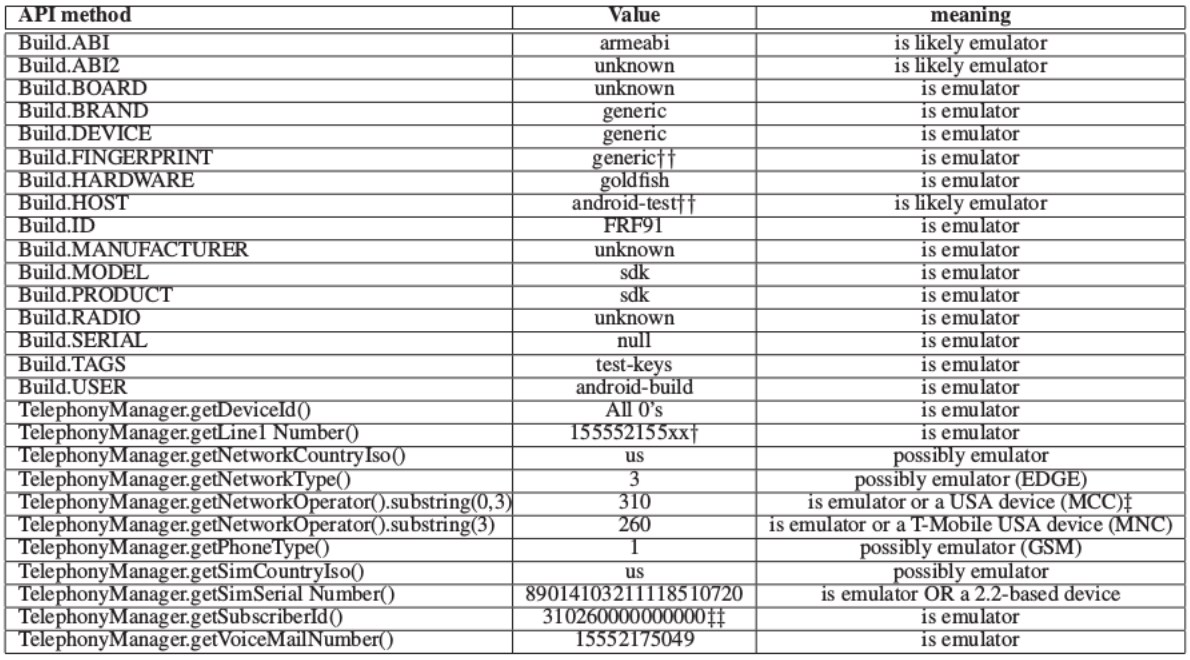
2.4.1 Evading Dynamic Analysis via Sandbox Detection

This topic will cover several techniques that can be employed to detect dynamic analysis systems in Android. These techniques detect differences based in: behavior, performance, hardware and software components. Furthermore, they require minimal or no privileges, so they can be invoked from normal applications of the market.

Behavioral differences

Android API

Behavioral differences are found using the API that Android provides. This topic will give some examples of what can be done with Android API to detect if the malware is running on a device or on an emulator. One typical example of what a malware writer can do is to extract the IMEI number (through *android.telephony.TelephonyManager.getDeviceId()*). Every Android device should return its IMEI, however, an emulator returns all 0's. The following image shows more examples of the potential of the Android API for detection: Some of the previous methods instantly detect the emulation environment, while others might need a combination of other factors to successfully detect the emulated environment. Two examples of that are the Mobile Country Code (MCC) and the Mobile Network Code (MNC) values which return values associated with T-Mobile USA on the emulator. However, real devices can actually output those values as well. If one of the both returns another value, it means that the returning values might being spoofed by the emulator.



API method	Value	meaning
Build.ABI	armeabi	is likely emulator
Build.ABI2	unknown	is likely emulator
Build.BOARD	unknown	is emulator
Build.BRAND	generic	is emulator
Build.DEVICE	generic	is emulator
Build.FINGERPRINT	generic††	is emulator
Build.HARDWARE	goldfish	is emulator
Build.HOST	android-test††	is likely emulator
Build.ID	FRF91	is emulator
Build.MANUFACTURER	unknown	is emulator
Build.MODEL	sdk	is emulator
Build.PRODUCT	sdk	is emulator
Build.RADIO	unknown	is emulator
Build.SERIAL	null	is emulator
Build.TAGS	test-keys	is emulator
Build.USER	android-build	is emulator
TelephonyManager.getDeviceId()	All 0's	is emulator
TelephonyManager.getLine1 Number()	155552155xx†	is emulator
TelephonyManager.getNetworkCountryIso()	us	possibly emulator
TelephonyManager.getNetworkType()	3	possibly emulator (EDGE)
TelephonyManager.getNetworkOperator().substring(0,3)	310	is emulator or a USA device (MCC)‡
TelephonyManager.getNetworkOperator().substring(3)	260	is emulator or a T-Mobile USA device (MNC)
TelephonyManager.getPhoneType()	1	possibly emulator (GSM)
TelephonyManager.getSimCountryIso()	us	possibly emulator
TelephonyManager.getSimSerial Number()	89014103211118510720	is emulator OR a 2.2-based device
TelephonyManager.getSubscriberId()	310260000000000‡‡	is emulator
TelephonyManager.getVoiceMailNumber()	15552175049	is emulator

Figure 2.7: List of API methods employed for emulator detection [T. Vidas, 2014]

Emulated networking
The emulator has a quite different networking configuration, with respect to real smartphones. The emulated network always reserves the 10.0.0.2/24 address space. Additionally, DNS resolvers will always assign the same address to the emulator (1, 2, 3-6 and 15). It is possible that a real device has this same network configuration, but is highly unlikely. However, if this is the only check made, the assumption makes it safe to have a device connected to a network with that exact configuration[1].

Underlying emulator
To emulate Android devices, it is used the QEMU emulator [Bellard, 2005]. It is possible to detect the emulator by its virtual address allocation [Matenaar and Schulz, 2012] or the CPU bugs [T. Raffetseder and Kirda, 2007]. This approach, has however two drawbacks. The first one is the mandatory usage of native code, which is not commonly used in Android Apps, therefore any App that uses native code can be considered as suspicious. The second drawback is the need of root privileges to access the CPU. Android sandboxing architecture makes it really difficult to gain those privileges [T. Vidas and Christin, 2011], so it is better to rely on other methods.

[1]it is safe because we are assuming that the malware will hide its malicious behavior

Differences in performance

Using CPU emulators have an inherent performance penalty, since they need to translate instructions for execution. However, desktop CPUs are considerably better than the CPUs used in smartphones. Therefore it is worth testing if the performance of the emulator is equivalent to the performance of the smartphones themselves. A performance benchmark [1] taken by the researchers [T. Vidas, 2014], proves that the performance penalty caused by the emulators is enough to distinguish the running environment (an emulator or a real device).

Table 2.2: CPU benchmarks

Device	**Average Round Duration (Seconds)**	**Standard Deviation**
PC (Linux)	0.153	0.012
Galaxy Nexus (4.2.2)	16.798	0.419
Emulator 4.2.2	68.872	0.804

On a different perspective, the graphical performance can also be tested. The results obtained on the graphical performance tests show [T. Vidas, 2014] that the physical devices are more consistent (few variations of the FPS[2]) and have high frame rates (between 57 and 60 FPS). Conversely, the same case does not happen with emulators that, regardless the Operating System, are either inconsistent or have low frame rates. Thus, it is possible to create an heuristic such that, by measuring the frame rate and continuously sampling the FPS values, it is possible to infer whether it is running on a physical device or on an emulator. More precisely, if a frame rate, v, is higher than 30 FPS and 80% of the samples are within the interval v-1 and $v+1$, then it is running on a physical device.

[1] Pi calculation round duration on tested devices using AGM technique (16 rounds)

[2] frames per second

Differences in components

Hardware components

There are several values for hardware components on Android. One good example is the CPU serial number that, as it is in our case, ARM CPUs return a unique string. However, CPU emulators always show sixteen zeros. Likewise, it is also possible to retrieve current CPU frequencies on a real device, contrarily to the 4.2.2 emulator, for instance. Lastly, the sensors are a good way to distinguish a smartphone from an emulator. A way to do this is by getting the number of sensors that the device supports. This is a naive approach since the SDK can be easily modified to return the expected value. To address this, it is possible to inspect sensor by sensor. For instance, the Android emulator return always the same values for the accelerometer sensor (0.0, 9.77622, 0.813417), which is quite different from its usual behavior.

Software components relating to Hardware

Verifying the battery status over time can be used to detect an emulated environment. Contrarily to a real device, the emulator usually keeps its battery status at the same level. Furthermore, vendors often add special hardware to the smartphones or implement kernel modules to use hardware not supported by Android. It is possible to exploit this by requesting the list of the kernel modules. This approach can, obviously be more complex if one wants to inspect modules themselves.

Software components

At the Software components level, it is possible to check which applications, that are usually installed in all Android devices, are not present. Examples of those applications are: Google Play, Google Maps API and Google Talk. One way that one can use to perform these checks is by using Android API to query the installed applications. Lastly, as the emulator does not have Google Services Framework, it is possible to query it. If the return value is *null*, then the malware is running inside an emulated environment.

Differences due to system design
The number of malware samples is growing everyday. Security experts must take some decisions like the timespan that each malware will run inside the emulator (in order to be analyzed). This timespan cannot be too high, since it will be nearly impossible to process every malware that way. So, malware can circumvent analysis if it starts its malicious activities after the timespan defined by the security expert. In case of Android, malware writers typically inspect the data stored in the smartphone (like messages, contacts, installed applications) and uses a heuristic to verify if it will execute its malicious code or not.

2.4.2 Anti-analysis Techniques

Malicious Android apps apply anti-analysis techniques to evade detection. According to one study [T. Petsas and Ioannidis, 2014], these techniques can fall into three distinct categories as follows:

- *static heuristics*, which analyze fixed values in the emulated environment;
- *dynamic heuristics*, that focus its analysis on the sensor's behaviors;
- *hypervisor heuristics*, which focus its analysis on the lack of emulated hardware

Static Heuristics
Static heuristics detect emulated environments by analyzing unique identifiers, such as the device ID, the current build version, or the layout of the routing table. The *device ID* makes it possible to uniquely identify a smartphone using the IMEI (International Mobile Station Equipment Identity) or the IMSI (International Mobile Subscriber Identity). The first one, is an unique number that identifies a device in the GSM (Global System for Mobile Communications) network. The IMEI is indeed applied by malicious Android Apps to evade out-of-the-box analysis [VRT, 2013]. The second one identifies the SIM card of smartphones. Static heuristics are the most simple heuristics used to evade analysis. Even though relatively easily to overcome, these techniques are proven to be effective [T. Petsas and Ioannidis, 2014; T. Vidas, 2014]. In order to perform these checks, one needs to require the

READ_PHONE_STATE permission in the Android Manifest file.

The *current build* also leaks information about the kind of environment where the Android App is executing, because it is extracted from the system properties. This information is gathered using the Android API which supplies a class *Build* that has the fields *PRODUCT*, *MODEL* and *HARDWARE* that can be used to detect the execution environment.

The *routing table* refers to the static network configuration of the virtual router which provides address space within 10.0.2/24. By default the emulated network interface has the IP address 10.0.2.15. Once again it is possible to use these fixed values to detect the kind of environment.

Dynamic Heuristics
Dynamic sensor information, as it name suggests, refers to the analysis of the output of the sensors. Besides the hardness to recreate faithful output for each sensor, the number of sensors is growing, therefore it is a good differentiator factor between emulators and physical devices. Android emulator is quite limited in the device in terms of sensors. As it by default cannot simulate sensor traces, the way that that data can be generated is by using other sensor simulators [OpenIntents, 2007]. Current sensor's simulation of Android emulator is naive, generating always the same data, at equal sample intervals, or with some negligible standard deviation. Furthermore, not all the sensors that exist in the physical device are emulated by virtual environment. A possible way to detect the emulated environment is by checking whether the collected values and the sampling intervals are the same or not. This can be applied to every sensor of the device. There are some works that show these techniques employed to a reasonable set of sensors [T. Petsas and Ioannidis, 2014].

Hypervisor Heuristics
Scheduling on QEMU does not work in the same way as it does on physical de-

vices. While in a physical device the scheduling can occur at any time, on QEMU it typically occurs after a basic block execution. This happens because the virtual program counter (PC) is not update at every instruction execution. Updating the PC would require another instruction, so deleting that instruction will improve performance. However, since the PC is not being updated at the basic block execution time, it is impossible to know for sure its value. Therefore, the scheduling events can only be treated when the PC gets updated again (at the end of the block). A technique that exploits is already implemented [Matenaar and Schulz, 2012].

The second technique that can be employed relies on self-modifying native code [Bramley, 2010]. On ARM-based smartphones, the cache does not guarantee consistency. Thus, it is possible to execute old instructions, since in some cases they must be explicitly invalidated. A code snippet that changes itself might lead to the execution of the cached code. On the other hand, as QEMU discards the previous blocks and then retranslates the newly generated code, as it tracks code pages modifications. Hence, QEMU does not have the risk of executing old instructions. As the behavior of the CPU and of the emulator difer, it is possible to infer the executing environment. To implement and use these two heuristics, a new folder named *lib* needs to be created inside the top level directory with the desired native code in the form of shared libraries. Additionally, the virtualized environment should support JNI.

Countermeasures

Hereby it will be present some countermeasures that could be employed to enrich analysis and prevents malware evasion.

At the static level, it is possible to easily modify the Android emulator in order to be resilient to the proposed static heuristics. QEMU enables the configuration of IMEI and IMSI so that these techniques will basically fail. Additionally, as for the build checking, it is possible to change the build.prop file that sets that value. Similarly to the other static heuristics, network configuration can also be changed as other tools [1] do.

Dynamic heuristics rely on sensors to evade analysis. The solution is to support most or all of the existing sensors. Furthermore, sensors must reproduce faithful outputs based on realistic user interactions, for instance. In order to attain this, external simulators might be used [OpenIntents, 2007]. This is no easy task and needs a deeper analysis.

The scheduling QEMU issue can be addressed by always updating the virtual program counter of the emulator. This will, however, produce a performance penalty that leads to an easier detection technique based on performance comparison.

Another prevention technique that could be employed is Hardware-Assisted Virtualization. In the ARM case, it is possible to use their virtualization technologies [ARM®, 2010]. This way, the virtualized environment will be easier to build and more suited for the processor. The substitution of QEMU for this technology would prevent any hypervisor heuristic.

The most robust solution is the development of an hybrid application execution, that is, use the previous solutions for the emulator and only allow the native code execution on a real device. This way, it could not detect that the malware is being analyzed.

2.5 Conclusions

2.5.1 Problem Statement

This chapter described the existing tools which analyze the software behaviors. As referred in the subsection 2.2 of this chapter, most modern tools rely on emulated environments to perform analysis. Dynamic analysis can be performed through a sophisticated inspection of the system calls and its parameters or by performing a

[1]APK Analyzer: http://www.apk-analyzer.net/

two-level VMI. However, as the analysis relies on VMI, it inherits the disadvantages of using an emulated runtime environment, that is, the possibility of detection and evasion of such systems. There were several methods depicted in this chapter for detecting its execution on emulated environments, hence enabling the possibility to evade them.

2.5.2 Goals

Existing evasion techniques rely on the assumption that the analysis is VMI-based and, thus that it is performed using an emulated environment. Therefore, we propose an hardware-supported dynamic analysis tool which performs hardware-based memory introspection with the aid of a physical device which is able to debug and introspect the target's memory. We believe that by using a real environment to perform the analysis it will be harder for the malware writters to evade the analysis. In practice, real devices have more features like IMEI, network configurations and sensors, that QEMU does not support at all or supports with naive outputs (e.g. sensors may output nearly the same values).

Chapter 3

Background on Debugging and Tracing Solutions

The popularity of embedded devices encouraged the provision of good tracing and debugging technologies. Tracing solutions automatically log informations about a target's execution. Targets might be software programs, for instance, or a piece of electronic device like a System-On-Chip (SoC). The recorded information is mainly used as a debugging aid in order to find and eradicate parts of a target where it is misbehaving with respect to its original design. These program faults are vulgarly known as bugs or vulnerabilities.

Tracing and debugging solutions might explore two ways of getting information of the target system: intrusive, non-intrusive. Debugging techniques are intrusive because they change the program's flow on the developers' will, through breakpoints, watchpoints, or by halting the system. As for the tracing techniques, it is possible to classify them as intrusive and non-intrusive, as well, but in a different perspective. Tracing is a technique that aims to act transparently enough so that it does not influence the system's execution, which is a requirement for real-time systems, for instance. However, tracing techniques might also be intrusive when they insert code on applications - software tracing. The second tracing method (hardware tracing) collects required information at run-time transparently for the software, at regular intervals for further analysis. Considering the main goal of this thesis - collecting runtime information from a running process of a Linux-based

OS to reconstruct runtime events -, it is mandatory to inspect these solutions and conclude with which ones it is possible to gather and inspect data from system calls.

This chapter contains the analysis of the available technologies for debugging ARM architecture, since as aforementioned it is the mostly used architecture on smartphones [ARM®, 2015].

3.1 ARM Debugging and Tracing

The debugging process is composed by three elements that communicate with each other:

- Debug host;
- Protocol converter;
- Debug target

The debug host is a computer, running a software debugger such as Open on-chip debugger (OpenOCD [Rath, 2005]). It is possible, from the debug host to issue high-level commands such as setting a breakpoint at a certain address or examining the registers' values at some point of the program's execution. The debug host connects to the target using an interface like JTAG. The target is typically a system with an ARM-based processor, like Cortex-A9 processor.

3.1.1 CoreSight

Embedded Systems debugging is quite different from desktop debugging since they have fewer components (they lack keyboard, monitor, etc). They are also harder to debug since it might involve testing the hardware components themselves. Additionally, multicore increased their complexity which enhanced the complexity of provide stable systems. Such problems coupled with the increasingly use of ARM processor-based SoCs made it crucial to provide good debug and trace solutions. It is essential for SoCs designers to provide systems that work well. In this context, ARM decided to create CoreSight IP, which is an on-chip debug and real-time trace solution for the entire SoC. This solution provides the following capabilities for system-wide trace:

- Debug and trace visibility of the whole system;
- Cross triggering support between SoC subsystems;
- Multi-source trace in a single stream;
- Standard programmer's models for standard tools support;

CoreSight is meant to be generic enough to provide many different components, so it can be possible for the SoCs designers to define the set of functions provided for debugging and tracing. This flexibility is quite important since the integration of such components has a cost attached. Additionally, different ARM processor-based architectures can have slightly different CoreSight specifications. Exactly for that reason, it is better to focus on the architecture used for this work. As one can see in the chapter 4, the SoC that we are using is OMAP4460 - an ARM Cortex-A9-based processor. The processor also adheres to the ARMv7 architecture.

The following topics will describe the CoreSight Design Kit for Cortex-A9 series processors **??**), since it will be the one adopted to the SoC that is in our embedded device. Moreover, the following topics' purpose is to give an overview of the most important components for this work rather than explaining the whole architecture exhaustively.

3.1.2 Debug Access Port

The Debug Access Port (DAP) is an implementation of ARM Debug Interface (ADI), that is inherited by ARMv7 architecture. It allows debug access to the whole SoC using master ports. These master ports are from two categories: Debug Ports(DPs) and Access Ports(APs). While Debug Ports are used to access DAP from external debugger, Access Ports are used to access the on-chip system resources. In order to get access an control the components externally, one should use SWJ-DP (Serial Wire and JTAG Debug Port). The components that are seen and controlled afterwards are the following:

- AHB-AP (Advance High-performance Bus AP), which will grant access to the System Bus Access Port;
- APB-AP (Advanced Peripheral Bus AP), which will grant access to the Debug Bus Access Port and a block memory (ROM) through APB-Mux;
- JTAG-AP, which will grant access to JTAG scan chains.

The SWJ-DP allows two ways of connection: via Serial Wire Debugging; via JTAG dongle. Then, through an external hardware tool, for instance RealView, it is possible to communicate and perform operations to the DP. The following figure resumes this process:

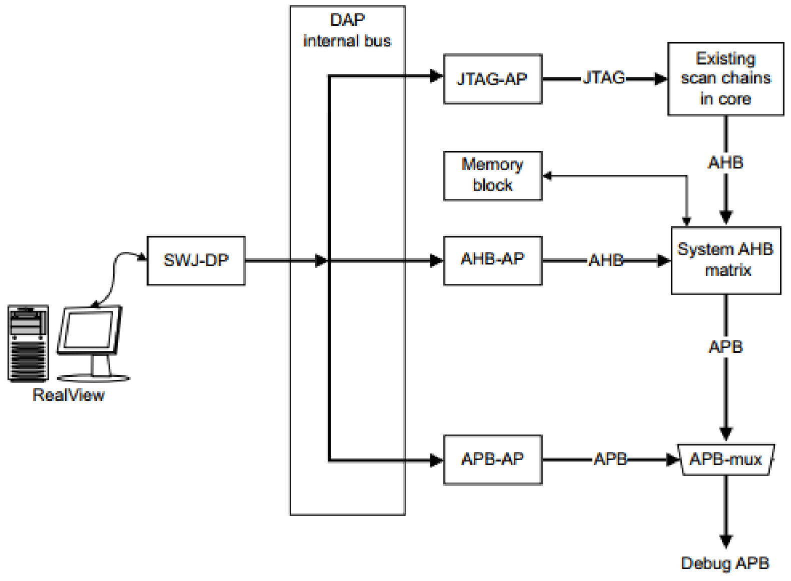


Figure 3.1: DAP Control Flow [ARM®, 2012]

3.1.3 Software Tracing

Software tracing is the cheapest and simplest way of tracing that CoreSight architecture provides. It generates data from software that is running on the cores themselves. That information is then written to the Coresight Instrumentation Trace Macrocell (ITM) which will stream data to the trace buffer. This form of tracing is known as SoftWare Instrumentation Trace (SWIT). Its main uses are:

- *printf* style debugging;
- Trace OS and application events;
- Emit diagnostic system information.

By analyzing its main uses it is possible to see that it is quite intrusive with respect to the insertion of code into the application. In order to output this traces it is needed to compile with some specific toolchain for ARM processors. Our aim is to debug without compile applications with some toolchain or debug flags, because malware can detect debugging environments [1], which represents a huge drawback for this work.

3.1.4 Program Flow Trace Macrocell

The Program Trace Macrocell (PTM) allows real-time instruction flow tracing based on the Program Flow Trace (PFT) architecture. The data recorded this way by trace tools can be used to reconstruct the execution of all or part of a program. The PFT architecture traces only at certain moments in the program's flow, called *waypoints*. PFT was designed this way because ETM protocol [ARM®InformationCenter, 2011] would generate big amounts of data that were not easy to process. Waypoints occur when there are changes of the program flow or events, such as an exception. Trace tools are able to get up to the following PTM traces:

- Indirect branches, with target address and condition code;
- Direct branches with only the condition code;
- Instruction barrier instructions;
- Exceptions, with an indication of where the exception occurred;
- Changes in processor instruction set state;
- Changes in processor security state;

[1]Anti-debugging and Anti-VM techniques and anti-emulation: http://resources.infosecinstitute.com/anti-debugging-and-anti-vm-techniques-and-anti-emulation/

- Context-ID changes;
- Entry to and return from Debug state when Halting Debug-mode is enabled.

Moreover, it is possible to configure the PTM to trace:

- Cycle count between traced waypoints;
- Global system timestamps;
- Target addresses for taken direct branches.

3.1.5 Embedded Trace Buffer

The embedded trace buffer (ETB) receive data from the Trace Bus (ATB), which might send it directly or not. It is a small on-chip component where trace information is stored. It contains the data recorded from the ETM. The buffer can be read through the JTAG port of the device once the capture has been completed. Hence, it is not required a special trace port. If the buffer is full, the information captured will overwrite the existing one.

3.1.6 Trace Port

The TPIU is the component where the trace hardware connects to in order to collect traces. Data that is being streamed from ATB will be sent directly to this component. This way, it is possible for the tracer to collect data continuously. It is possible that the data generated is more than the port can output. Under such scenario, the trace from ETM is sent accordingly to the capacity of the port.

3.1.7 Hardware Breakpoints

Hardware breakpoints are a type of breakpoints which is integrated into the SoC. In general, breakpoints enable to stop the target's execution when the program reaches a certain address. As it is implemented in hardware, these kind of breakpoints are a set of programmable comparators that can be set with a specific address value. When the program address bus matches the previously set bits, a signal to halt the target's CPU is sent.

In Cortex-A9, a *breakpoint register pair*(BRP) must have its bits programmed in order to set a breakpoint. Each BRP is composed by one breakpoint register control(BRC) and one breakpoint register value(BRV). While the BRV holds an address or a context ID, the BCR holds the possible options of the breakpoint triggering which are the following:

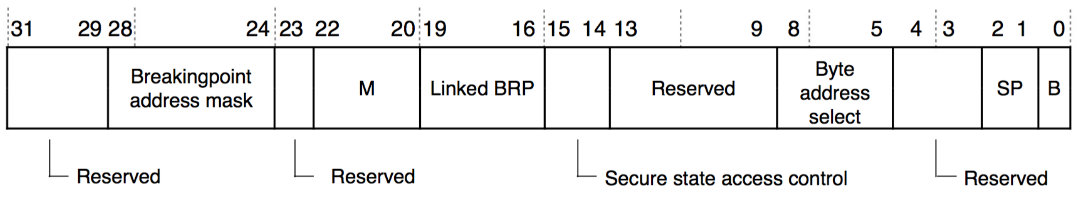


Figure 3.2: Breakpoint control register's bits

The previous register has many options which can be used, but I will explain how to set an hardware breakpoint, both on an address and on a context ID, because it will be used by the proposed tool on this work. As the BVR can only store either an address or a context ID, two BRP must be used and linked. The key factor lies on bits [20:22] and bits [16:19] of BCR, because they hold the meaning of the BVR value and the linked BRP number[1]. The meaning of the BVR is set to b011 on one of the BRP's, specifying that its BVR has a linked context ID while the other will have it set to b001, which means that it matches on a linked address. Each of the BRP's will have its linked BRP bits set to the other. The [5:8] bits are set to b1111 so it compares the 4 bytes of BVR. This setup ensures that the breakpoint only matches when both the address and the context ID are the same. This kind of breakpoints has a good performance since it is integrated on the hardware, but has a small number of breakpoints. For what matters the proposed tool, it is only needed one hardware breakpoint, so the limiting number being 6 is more than enough (even though only 2 of them support context ID comparison). Note that these kind of breakpoints have an additional required feature for this work: not traceable. In order these registers are only accessed with privileged mode or through JTAG.

[1]each BRP is designed to an id, for instance, BRP0

3.2 JTAG for ARM

In this section the reader will firstly read about JTAG and its facilities. Afterwards, there are lists of the JTAG debuggers, hardware tracers and software debuggers available in the market. For the sake of simplicity, the term *dongle* will be employed whenever it is needed to use a generic word for JTAG debuggers and hardware tracers, i.e., *dongle* is a piece of hardware that debugs a target and communicates with the host computer. In order to choose some specific *dongle*, it was fundamental a list of features. The selected features are the following:

Features	**Importance**
Support ARM processors	✓
Ability to inspect memory	✓
Support interrupts	✓
Someone has reportedly used it successfully	✓
Support GDBServer	✗
Support adaptive clocking (RTCK)	✗

Table 3.1: JTAG constraints

Legend		
	✓	Mandatory feature
	✗	Not mandatory

As one can check on the previous table, there were six constraints with two importance levels (mandatory feature,not mandatory). As for the level of importance, it is used ”mandatory feature” if the feature is crucial, and ”not mandatory” if the feature represents a help but it is not essential for the objective of this work. For the sake of clarity, features will be explained in the following paragraph.

Support of ARM processors is an obvious mandatory feature, since we want to debug a SoC that look the most with the ones present on smartphones (which are mostly ARM-based). The ability to inspect memory should be available on all the dongles already. Supporting interrupts is important as well, because it will be with the interrupts that we will try to reconstruct the system calls. In order to avoid some problems that might occur at configuration /development time, the selected dongle is one that had reportedly been used successfully. The way that this was verified was by finding tutorials, as updated as possible, for each dongle - if we have an updated tutorial, it will be less likely that we hit undesired bugs. The last two features are a plus since they are not really needed, i.e. they are a differentiating factor. GDBServer serves to debug the target remotely. It is possible to debug it with and without any compilation flag. Debugging an application without any compilation flag is harder, because the debugger does not have access to the code nor to its layout. GDBServer has also a monitor that lets it run JTAG commands remotely as well. The environment will only work properly if the dongle and the CPU target are synchronized. This constraint typically implies that the JTAG clock is changed to match the CPU clock. Adaptive clocking (RTCK) automates the process of changing the clock frequency of the dongle, so it adapts to the frequency used by the target CPU.

3.2.1 In-Circuit Emulator

Debugging of Embedded systems can be achieved with in-circuit emulators (ICE). In-circuit emulators are hardware devices that emulate the target CPU in order to add debugging facilities to it. In-circuit emulators are adapters that are inserted between the host computer (which is running a debugger software) and the target. This allowed to have a non-intrusive analysis of the program flow as well as to control it and to inspect CPU state, CPU registers and physical memory. However, costs of these equipments were getting prohibitively high because chips were getting faster which would require higher speed logic, hence more expensive adapters. This trend led vendors to provide better debug facilities to their chips. These facilities were then named as on-chip debug circuit.

3.2.2 JTAG capabilities

Joint Test Action Group (JTAG) stands for the IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture. Nowadays, systems use the target system's CPU directly, with special JTAG-based debug access, which are low cost solutions with respect to in-circuit emulators. Actually, in-circuit emulator has then extended its definition to include JTAG based hardware debuggers as well, even though they are not the same thing. In fact, instead of emulating the target, JTAG hardware debuggers leverage on-chip debug (OCD) capabilities to debug targets - which is an advantage since they are able to debug the target itself. On-chip debug circuit is the target's architecture for debugging and tracing purposes. A downside of this approach is that the debugger is tightly connected to the target's architecture, hence connected to and limited by its features.

JTAG is a standard that was designed to assist with device, board, and system testing, diagnosis, and fault isolation. It is an essential way of debugging embedded systems, since it can access to the sub-blocks of integrated circuits (ICs). It is possible to debug the wiring of the embedded system through boundary scan testing. Generally, smartphones' processors do not supply another way of debugging. JTAG enables the possibility of debugging even the early boot software which runs before anything is set up. Many silicon vendors provided new architectures, like ARM CoreSight, that enabled software debug, instruction tracing, and data tracing around the JTAG protocol.

Debugging

JTAG is widely used for IC debug ports. Embedded systems development relies on debuggers communicating with chips via JTAG to perform operations that act on the processor. Processors can be halted, single stepped, or run freely. One can set code breakpoints for code in RAM (often using a special machine instruction), in ROM/flash or set data breakpoints. It is also possible to use ARM Program Trace Macrocell to trigger debugger (or tracing) activity on complex hardware events.

3.2.3 JTAG Debuggers

JTAG Debuggers, also known as JTAG adapters, can access to the target processor's on-chip debug modules through JTAG protocol. Those modules grant access to the debug of an embedded system directly at the machine instruction level. This topic contains a small description about the table of JTAG debuggers **??**, which helped to analyze the market of these debuggers. The features were modified, with respect to the ones present in the table 3.1 to enlarge the number features that we could compare. Choosing the JTAG debugger was no easy task. In most of the cases it was not clear the difference amongst each debugger and the prices are quite different. To make things even worse, some of the vendors may request additional information in order to get access to some specifications. It was a time consuming task. In the end, we have chosen the Flyswatter 2, because it was mandatory to use a cheap solution and, among all of them, the flyswatter2 was the one that met the requirements of table 3.1. On top of that, this JTAG debugger has a good clock frequency (upto 30MHz) for its price.

3.2.4 Trace Hardware

Trace hardware may have the same features that JTAG-based probes have. On top of that, these dongles leverage the open-chip debugger capabilities of the target's processor to trace applications in a completely non intrusive way, that is, without getting changing the application's code (to include printf's, for example) or changing the program execution flow. In the ARM example, trace hardware will get access to one or all the ETM modules through the trace port interface unit (TPIU). This port accepts trace from the trace bus (ATB) directly or through a trace funnel, which may send the trace data to other component like ETB.

Trace hardware is too expensive for this work and, thus, it would not fit our cost-effective necessities. Additionally, it is possible to collect the same traces from ETB, which achievable with JTAG Debuggers. Even in this unfavorable scenario, these solutions were also analyzed. The table **??** shows the units analyzed in this work. As with the JTAG debuggers, the search was based on the requirements referred in the table 3.1. Furthermore, every evaluated tracer supports 1.8V (voltage of OMAP4460). When a tracer supports ARMv7 but does not support explicitly Cortex-A9, it is written that it supports ARMv7. Note that the table has more features than the ones shown in the requirements table. That is because we considered more features in order to compare between tracers.

3.2.5 Software Debuggers

A Software debugger is installed on the host computer and used to communicate with the target through the *dongle*. The features of some dongle and SoC might be limited by the software and vice versa. Depending on its features, the software can be the most expensive part of our system.
Some examples of software debugger are: Chameleon Debugger, MULTI IDE, SourcePoint for ARM and IAR embedded workbench and OpenOCD. Over the all possibilities, the only one which is opensource is OpenOCD. Although it lacks some features like multicore debugging, it is quite complete.

3.3 Conclusions

In this chapter several existing debugging and tracing solutions were analyzed. It is the intent of this work to employ such techniques to create an hardware-supported dynamic analysis environment avoiding the use of VMI-based techniques. Moreover, our solution must not have access to the source code of the Application nor insert debugging flags. Traditional debugging techniques are then excluded from our approach, because they need access to the source code of an Application in order to properly debug it (e.g. set breakpoints). In the same line of reasoning, software tracing techniques are not suited for this work, because they perform *printf* debugging by including code on the source code of an Application. Finally, the only solutions that fit this approach are hardware tracing techniques and hardware breakpoints, because they do not change neither the source code of a Linux program nor any component of the OS itself. The aim of this work to inspect memory to reconstruct system calls and respective parameters. Using CoreSight for that is too hard in the sense that there are some components (PTM, Cross Trigger Interface, TMC) which need configuration that are not implemented or have a less reliable implementation(obsolete code) on OpenOCD. On top of that it would be needed to get knowledge about proprietary formats of the information that can be read from ETB. Given this time-consuming tasks, I selected the hardware breakpoints, because they already work on the OpenOCD.
After the analysis of the existing solutions it was possible to infer that JTAG

debuggers and hardware tracers can both have access to the tracing mechanisms provided by the on-chip tracing and debugging techniques. The trade-off between these technologies is the ratio between their trace speed and price. For this work, the choice of this technology was biased towards the cost-effective solutions, so it will be used a JTAG debugger. The final choice was the Flyswatter2 because it is one of the cheapest debuggers analyzed that has a decent frequency of 30MHz.

Chapter 4

Approach

4.1 Approach Overview

This section intends to explain the approach used on OpenST(Open SiliTracer), which is the tool proposed on this work.
OpenST represents the very first effort taken on tracing runtime events by performing a fine-grained hardware-based memory introspection. OpenST leverages the capabilities of external hardware(subsection 3.2.2) and the capabilities of existing hardware debugging facilities(subsection 3.1.7) to analyze the memory of the target at runtime. This feature makes it possible to faithfully reconstruct sys-

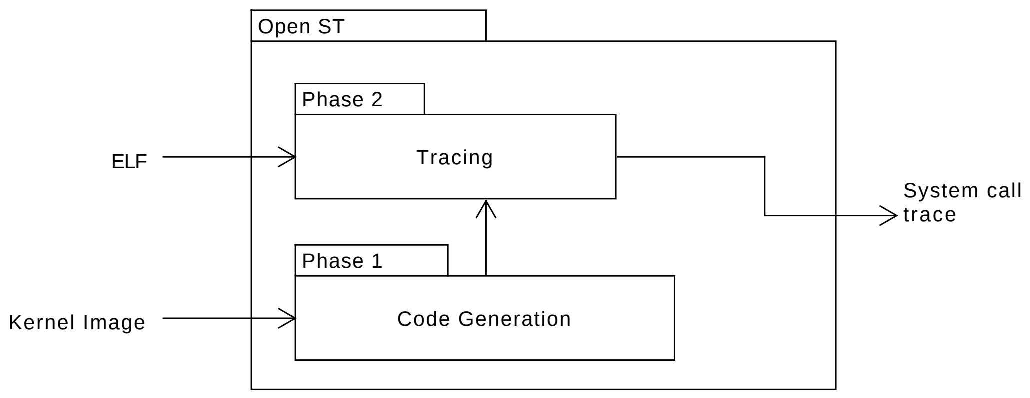


Figure 4.1: Open SiliTracer

tem calls - and their respective parameters - issued by a Linux program relying on hardware and not on hypervisors as the existing techniques do [K. Tam and Cavallaro, 2015; Yan and Yin, 2012]. Therefore, it is a technique which acts transparently to that target in the sense that it does not instrument its environment.

In order to get a better overview, check figure 4.1 that characterizes the high-level approach employed on OpenST. The first phase(**Code Generation**) is characterized by the code generation based on a kernel image. The output of that phase is a library to be used on tracing phase(**Tracing**). As the kernel image remains the same on the target, the first phase just needs to be executed once. On the other hand, the second phase(**Tracing**) may be executed any number of times. Each execution receives a Linux program as an input and traces its execution on the target at runtime. Finally it logs the traced data as depicted in section 4.3.5. Throughout the following sections, the reader will get a better understanding of each component of this approach.

4.2 Phase 1: Code Generation

The code generation phase can itself be divided in into two other separate phases as shown in figure 4.2. At the first phase, (**phase 1.1: System Call Data Reconstruction**), OpenST reconstructs the data needed from the system calls present in the kernel image. This information is, then, sent to the (**phase 1.2: Introspection Procedure Generation**) where the tool generates the introspect procedures, based on that received information, to be included as a library in the second phase(**Tracing**).

4.2.1 Phase 1.1: System Call Data Reconstruction

This sub-phase refers to the collection of all the information needed of the kernel image. Overall, the goal of this phase is to gather information about the system call prototypes and about the kernel structures and respective layouts in memory from the kernel image. In order to reconstruct the system calls, one needs to firstly know every parameter of it. Secondly, it is important to know the layout of every struct, typedef and union, since it will be with that information that the code can be generated in the next phase. The concrete output of this phase are two C generated files that are sent to the **phase 1.2: introspection procedure generation**.

4.2.2 Phase 1.2: Introspection Procedure Generation

Introspection procedure generation phase is where the memory dumping functions are generated. This phase receives the two previously generated files from the last phase. Holding the information about the system calls prototypes and about the structures and other non-basic types, it is possible to parse the parameters of the system call and apply a reflection technique on the most complex parameters to understand its inner declarations. Moreover, as this phase receives the memory layout of each non-basic data type, it is possible to generate the introspection procedures for each system call and data type. The output of this phase is the C library to be included in the next phase (**Tracing**).

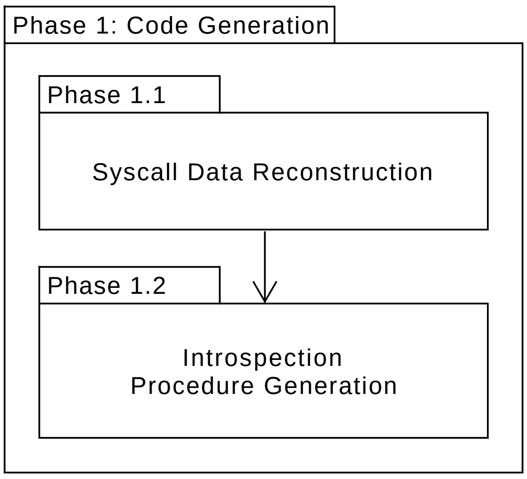


Figure 4.2: Phase 1

4.3 Phase 2: Tracing

The tracing phase, as previously stated, can be executed any number of times. In the following subsections I will explain all the components of this phase, depicted in figure 4.3. In general, this phase traces the issued system calls by an executing Linux program and logs that information to the user. In the first phase, OpenST sets a hardware breakpoint so that the target halts its CPU when it issues a system call. Then, when the target is halted, the system call tapping triggers the components needed to reconstruct the process data structures and the system call's parameters by means of memory introspection. Finally, when it holds all the data needed, it logs to the user and to a file in disk for further analysis.

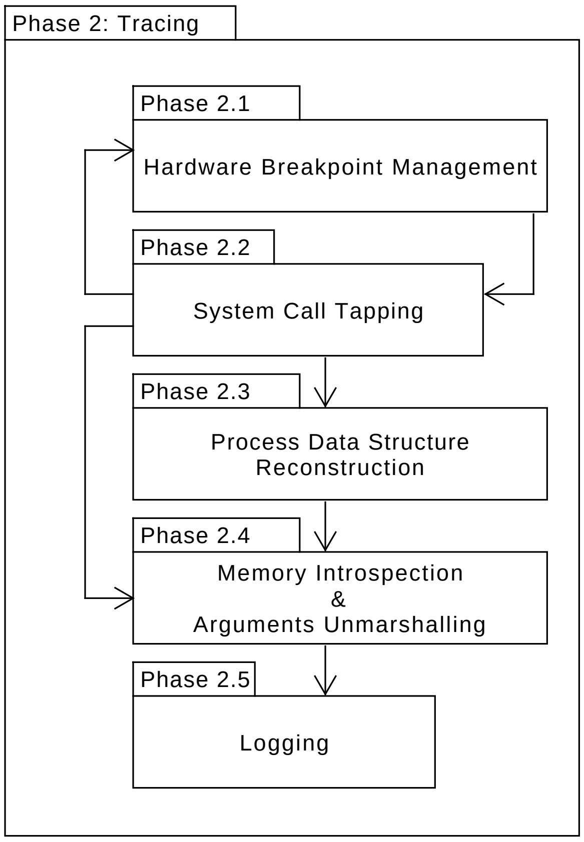

Phase 2: Tracing
Phase 2.1
Hardware Breakpoint Management
Phase 2.2
System Call Tapping
Phase 2.3
Process Data Structure Reconstruction
Phase 2.4
Memory Introspection & Arguments Unmarshalling
Phase 2.5
Logging

Figure 4.3: Phase 2

4.3.1 Phase 2.1: Hardware Breakpoint Management

This Management system deals with the hardware breakpoints' insertion so that the target's CPU halts its execution when the breakpoint's conditions are met. The conditions are, for this approach, the execution flow arrival at a specific address(*software interrupt handler*) and that the process identification of the issued system call is the one which is being traced. Additionally, this component unsets the breakpoints so that the target does not halt right after its execution resuming. To address this issue, a new breakpoint is set at the next word's address and the older breakpoint is unset. After arriving at the next word's address, the same process is employed and the current breakpoint is unset a the old breakpoint is reseted. This way, OpenST simulates a stepping mechanism.

4.3.2 Phase 2.2: System Call Tapping

System call tapping is a component that waits for the target to reach specific conditions set by the hardware breakpoint management. When the conditions are met, this component receives the information that the CPU is halted due to a breakpoint and calls the components responsible for reconstructing the system call and its arguments.

4.3.3 Phase 2.3: Process Data Structure Reconstruction

Process data structure reconstruction is the phase where OpenST reads the process-specific information from the memory and the registers. The collected information on this phase is the process identification(*pid*), the thread group identification(*tgid*) and the executable name(*comm*). These informations are stored in thread information structure, which can be accessed through the stack pointer(SP) register. It is worth noting that every process should be able to access this information.

4.3.4 Phase 2.4: Memory Introspection & Argument Unmarshalling

This is the phase where the previously generated library code is used. The intent of this code is to introspect registers and memory to generate the system call arguments. As some of the arguments may be more complex(pointers, non-basic data types), a depth level is introduced to OpenST. This depth level limits the level of inspection of the system call parameters. Therefore, if a given system call has a pointer to integer as a parameter, and the depth level is set to 0, it will print the integers' address. However, if the depth level is set to 1, it will print the integer itself. In this basic example, if the depth level was higher than 1, it should behave in the same way. The same line reasoning can be applied to complex structures, which can have inner declarations which are pointers to other structs, or to the same struct. In other words, it performs a recursive approach of inspecting the parameters receiving a depth level as a limit.

4.3.5 Phase 2.5: Logging

After gathering all the information related to the system call and respective arguments, OpenST just logs the information to the standard output and for a newly created file with the sample name. The latter log can be accessed afterwards, thus, permitting further analysis.

4.4 Phase 2: Conclusions

OpenST is a dynamic analysis automated tool, which allows perform a fine or coarse-grained analysis of the system calls and its parameters depending on the analyst needs. The novelty of this approach relies on this variable-depth introspection of the memory, since it can understand the contents of the memory and reconstructs them. By performing this reconstruction on the hardware level, can make that it is less likely for executing application to understand the difference between this new analysis environment and non-analyzed smartphone. The reason for this to happen is because the existing evasion techniques assume that the analyses environments are built on top of emulators, so checks like reading the IMEI, network configurations or even sensors data will basically fail.

Chapter 5

Implementation Details

This chapter presents the implementation details of Open SiliTracer.
Every approach phase of Open SiliTracer described on chapter 4 had code changes and I will describe all of them in the following sections. In the first two sections it is possible to get information on how the code is generated(**phase 1: Code Generation**).. The third section will provide to the reader all the changes performed on OpenOCD(Open on-chip debugger) [Rath, 2005], thus on **phase 2: Tracing**.

Note: Due to limitations of the configuration file of the SoC(System-on-a-chip) used on this work 6.2.1, I changed the configuration file of the kernel image provided by linaro, so it would disable SMP (Symmetric Multiprocessing). This modification disabled the multi-core of the target board.

5.1 Phase 1: Code Generation

This section holds the required knowledge about the code generation to be inserted in **phase 2: Tracing**. The **phase 1: Code Generation** has clear issues because it is time-consuming to write the hardware introspection code needed for each system call present in the kernel image[1]. In fact, the automatically produced code of this phase has 12707 lines of code, which is a good indicator of how time-consuming it would be to write the code and to debug it. Moreover, it is hard to have a deep understanding of all the data types passed as parameters and their layout in memory.
In order to address the aforementioned issues, I found Pahole [pro, 2007] and pycparser [eliben, 2015]. To summarize, I am using pahole as an aid to retrieve the the system calls prototypes and the information of structs, typedefs and unions related to their members and organization in memory. On its side, pycparser plays an important role as well by taking the pahole's generated information and automatically generating all the system call dumps and respective parameters.
This approach, besides having the direct advantage of working in an automated way, ensures less implementation errors on the final code. The following subsections provide a deeper analysis of these tools.

5.1.1 Phase 1.1: System Call Data Reconstruction

This subsection provides the implementation details of this phase. It presents an overview of the tool used and the modifications performed to make it useful to OpenST.
Pahole [pro, 2007] is a tool that leverages the debugging information on standard formats like DWARF and shows the data structures layout in memory. The initial aim of this tool is to inspect structures and optimize its size in memory, since the ordering of the structures' members matters. However, it is still possible to gather information on other data types and on functions.

Regarding what concerns this phase, OpenST takes advantage of the previous

[1]There are 377 system calls in the target's architecture

features to generate two C files. One of them holds the information about the system calls' prototypes in the form shown in the snippet 5.4. The other file holds the information about the structures, typedefs and unions of the kernel. The only requirement of this tool is that the ELF (executable linked file) has been generated with the flag -g. In terms of kernel compilation, this means enabling CONFIG_DEBUG_INFO. The kernel source tree which I am using to compile the kernel has that configuration enabled by default.
In order to make pahole useful to this work, I made some modifications to this tool so it would output the information on a format that pycparser understood. It is possible to output the structs of the kernel with and without some comments with further information. It is of paramount importance to delete the comments from the final ouput, since pycparser does not understand comments. However, if the comments are deleted, pycparser will lose the data regarding the memory alignment as it is possible to check in the next image.

```c
struct thread_info {
  // ...
  struct task_struct *task;       /* 12 4 */
  struct exec_domain *exec_domain;   /* 16 4 */
  __u32 cpu;              /* 20 4 */
  __u32 cpu_domain;           /* 24 4 */
  struct cpu_context_save cpu_context; /* 28 48 */
  // ...
  __u8 used_cp[16];           /* 80 16 */
  long unsigned int tp_value;      /* 96 4 */
  struct crunch_state crunchstate;   /* 100 184 */

  /* XXX 4 bytes hole, try to pack */

  union fp_state fpstate;  /* 288 140 */

  /* size: 752, cachelines: 12, members: 16 */
  /* sum members: 740, holes: 3, sum holes: 12 */
  /* last cacheline: 48 bytes */
};
```

Listing 5.1: Pahole output before instrumentation

At the rightmost part of each structure member, there is the information about the offset and size, respectively, of that specific member. The modification taken on pahole refers to the switching of the comments by new struct members with the information on the offset and size of each struct member.

```
struct thread_info {
  // ...
  struct task_struct *task;
  int arm_tracing_offset[12];
  int arm_tracing_size[4];
  struct exec_domain *exec_domain;
  int arm_tracing_offset[16];
  int arm_tracing_size[4];
  __u32 cpu;
  int arm_tracing_offset[20];
  int arm_tracing_size[4];
  __u32 cpu_domain;
  int arm_tracing_offset[24];
  int arm_tracing_size[4];
  struct cpu_context_save cpu_context;
  int arm_tracing_offset[28];
  int arm_tracing_size[48];
  // ...
  __u8 used_cp[16];
  int arm_tracing_offset[80];
  int arm_tracing_size[16];
  long unsigned int tp_value;
  int arm_tracing_offset[96];
  int arm_tracing_size[4];
  struct crunch_state crunchstate;
  int arm_tracing_offset[100];
  int arm_tracing_size[184];
  union fp_state fpstate;
  int arm_tracing_offset[288];
  int arm_tracing_size[140];
};
```

Listing 5.2: Pahole output after instrumentation

The previous output is the one which is received by pycparser. Note that the offset and size have a prefix(arm_tracing_), to distinguish the metadata inserted from the real structure members. This metadata will be processed by pycparser in the next phase(**Introspection Procedure Generation**). Overall, the novelty introduced in pahole is the dumping of typedefs, functions, unions and enums.

Challenges

There were some challenges addressed during the instrumentation of pahole, which are the following:

- Uniqueness of data types;

- Ordering of data types;
- Limitations of pycparser

Throughout the default output, there were some data types repetitions which made the code more verbose. Additionally, there were dependencies from some *typedef* ed types to the others, that is, one typedef that it is dependent from another which, by default is found later in the file. The instrumentation part solved the latter one by adding a component which orders dependencies in the output. As for the code repetitions, there is now an hashmap which checks whether the data type is already in the output or not. Overall, pahole's issues were related with pycparser limitations, so the instrumentation of these tools was tightly coupled.

5.1.2 Phase 1.2: Introspection Procedure Generation

The implementation of **Phase 1.2: Introspection Procedure Generation** uses a C99 parser written in python, namely pycparser [eliben, 2015]. As explained in the phase **System Call Data Reconstruction**, this phase receives the system call prototypes and the memory alignment of structures, opaque types[1] and unions present on the kernel image. By the end of this phase, OpenST should have all the code, in form of a library, needed to perform hardware-based memory introspection. This subsection clarifies the details of the implementation, specifically the main changes performed on pycparser.
OpenST uses pycparser to generate the AST of the kernel structures and of the system call prototypes. Furthermore, it stores the metadata needed to understand the kernel structures, unions and opaque types. This metadata is the information of the unions' size, structures' members and respective offsets and sizes and the same information of opaque types that hide structs and unions. In order to avoid undefined references by *typedef* ed types passed as arguments, OpenST merges both C files, inserting the content of the structures on the top. At this point, pycparser generates the AST for the merged file and it starts visiting all the system call prototypes. For each system call prototype, this tool generates a system call definition and, depending on the parameters, it generates memory introspection code. Parameters can be basic or complex and, for that reason, this tool supports a depth level. The first level generates the dumping code directly from the registers. The next levels, if the parameter is not a basic type, are introspect the physical memory. The final introspection code aims to read the system call's parameters directly from memory. So, as the tool contains the required metadata regarding the offsets and the sizes, it is not needed to have the structs on the final code. Therefore, the last step is to remove the structs from the previously merged code.

[1]Opaque types refers to the typedefs

The following snippets show the inputs(listings 5.3 and 5.4) and the output of this phase.

```c
typedef long int __kernel_time_t;
typedef int __kernel_clockid_t;
typedef __kernel_clockid_t clockid_t;
struct timespec {
    __kernel_time_t tv_sec;
    int arm_tracing_offset[ 0];
    int arm_tracing_size[ 4];
    long int tv_nsec;
    int arm_tracing_offset[ 4];
    int arm_tracing_size[ 4];
};
```

Listing 5.3: Structures and typedefs used on clock_gettime

```c
long int sys_clock_gettime(clockid_t const which_clock, struct timespec * tp);
```

Listing 5.4: clock_gettime system call prototype

When the parser is generating the code and it reads the first parameter, it tries to translate that value in a complex data type or in a basic data type. Therefore, in this case, *clock_id* is decoded to *int*.

```c
char *dump_sys_clock_gettime(int depth, struct target *target)
{
  char **dumped_params;
  char *param_str;
  int len = 0;
  if (depth < 0)
  {
    param_str = malloc(0);
    return param_str;
  }

  unsigned int arm_tracing_which_clock = get_uint32_t_register_by_name(target->reg_cache,"r0");
  unsigned int arm_tracing_tp = get_uint32_t_register_by_name(target->reg_cache,"r1");
  dumped_params = malloc(2 * (sizeof(char *)));
  if (depth == 0)
  {
    len += dump_int(arm_tracing_which_clock, &dumped_params[0]);
    len += dump_ptr(arm_tracing_tp, &dumped_params[1]);
    param_str = copy_params(dumped_params, 2, &len);
    free_dumped_params(dumped_params, 2);
    return param_str;
  }
  if (depth >= 1)
  {
    len += dump_int(arm_tracing_which_clock, &dumped_params[0]);
    len += dump_timespec(depth-1, arm_tracing_tp, &dumped_params[1], target);
  }

  param_str = copy_params(dumped_params, 2, &len);
  free_dumped_params(dumped_params, 2);
  return param_str;
}
```

Listing 5.5: Memory introspection code of clock_gettime

The code organization of **Phase 2: Tracing**(section 5.2) influences generated code, since it expects that every system call dumping function returns a string(*param_str*) with the format of the dumped parameters, that is, "*(arg1, arg2, ..., argn)*". Overall, the dumping code starts by reading the parameters from the registers, and then it enters in the depth level block. In this block, the contents of the parameters are written to the auxiliary variable, *dumped_params*, depending on the depth level, dereferencing, in between the *if* statements, the arguments if needed. Right after returning, it writes the contents of *dumped_params* to *param_str*.

```c
int dump_int(unsigned int value, char **param_str)
{
  int len = dump_generic(param_str, NUM_CHARS_INT, "%d", value);
  return len;
}
```

Listing 5.6: Dump integer from register

Dumping the value from the register follows the same principle, whether the value is a pointer, an integer, or another data type. The only thing that changes is the number of characters of the string and the format specifier, which would be *0x%x* for the pointer. An additional placeholder which worths being referred is the one used on *enum*, because it may not be intuitive that it can be dumped as an integer.

```c
int dump_long_int_from_mem(unsigned int addr, char **param_str, struct target *target)
{
  unsigned int *value = get_address_value(target, addr, SIZE_OF_LONG);
  int snprintf_n_read = dump_generic(param_str, NUM_CHARS_LONG, "%li", *value);
  free(value);
  return snprintf_n_read;
}
```

Listing 5.7: Dump long integer from memory

The previous snippet uses a procedure which is created in **Phase 2: Tracing**(section 5.2), *get_address_value* to read from memory the contents of an address with variable size. Then, the value read is written as a string on *dump_generic* procedure.

```c
int dump_generic(char **param_str, unsigned int size, char *format, unsigned int value)
{
  *param_str = malloc(size);
  int snprintf_n_read = snprintf(*param_str, size, format, value);
  return snprintf_n_read;
}
```

Listing 5.8: Dump generic type's value

The previous procedure just dumps a given value to a string and returns the number of bytes written. As shown in the dumping code of the system call, the written string may be merged with others in order to display the value in the required format, which is "*(arg1, arg2, ..., argn)*".

```c
int dump_timespec(int depth, unsigned int addr, char **dumped_params, struct target *target)
{
  char **dumped_type_params;
  unsigned int arm_tracing_tv_sec = addr;
  unsigned int arm_tracing_tv_nsec = addr+4;
  int len = 0;
  if (depth < 0)
  {
    *dumped_params = malloc(0);
    return len;
  }

  dumped_type_params = malloc(2 * (sizeof(char *)));
  if (depth >= 0)
  {
    len += dump_long_int_from_mem(arm_tracing_tv_sec, &dumped_type_params[0], target);
    len += dump_long_int_from_mem(arm_tracing_tv_nsec, &dumped_type_params[1], target);
  }

  *dumped_params = copy_params(dumped_type_params, 2, &len);
  free_dumped_params(dumped_type_params, 2);
  return len;
}
```

Listing 5.9: Dump structure timespec

The way that the dumping of a struct works is similar to the way that the system call works. The main difference is that the first level reads the contents from memory and not from registers. Note that the declarations of the variables which hold the addresses to be read have an offset ($addr + offset$) embedded in the code. That offset comes from the metadata previously stored.

Challenges

There were some challenges encountered throughout the instrumentation of pycparser. Hereby it is the three more important challenges:

- Recursive dumping of each structure;
- Translation of *typedef*ed data types;
- Metadata of anonymous structure

The complex data types, like structures need to have a recursive approach of generating the dumping functions. This happens because there may be the case where the structure has structures as its members, whether that structs are different from the original one or not. The translation of typedefs is time-consuming even though it applies a dynamic approach, storing intermediate values. The third problem was the hardest to deal with. In Linux kernel, it is possible (and gcc accepts) to create structs without a name inside other structs. This possibility made it harder to store a name for dumping that struct. The way our tools addresses this issue is by concatenating the type of each of ist structure members.

5.2 Phase 2: Tracing

This section describes how this phase performs analysis on an executable linked file(ELF). In order to do it, OpenST uses a open source tool that allows to handle all the JTAG communication between a host device and a target board running Linux. That tool is named Open On-Chip-Debugger(OpenOCD) [Rath, 2005] and hereby there is an explanation on how it is instrumented.

5.2.1 Phase 2.1: Hardware Breakpoint Management

In Linux ARM it is possible to invoke a system call by means of the *swi* instruction[1], which stands for software interrupt. The only purpose of this instruction is to make a system call to the operating system. After reading such instruction, the CPU generates a software interrupt exception and the control flow changes to *swi_vector*. This procedure is a wrapper function for every system call.
OpenST leverages the debugging capabilities of the CPU 3.1.7 to halt the target's execution when the control flow reaches the *swi_vector* address. In order to do it, OpenST inserts an hardware breakpoint on the *swi_vector*'s address[2]. When the CPU halts, the breakpoint must be removed, because otherwise it would halt the CPU right after the target resumed. OpenST addresses this issue by inserting a different breakpoint on a second address(*swi_vector*'s address + 4) and alternating between adding and removing the breakpoint on those addresses. This is the way it simulates the stepping mechanism:

[1]Nowadays, the term has changed to SVC (supervisor call). I employed the old term because the kernel version which I am using still uses the *swi* instruction

[2]One can know this information by the System.map file generated after the kernel compilation

```
1  /* pc_value holds the address of the current breakpoint */
   breakpoint_p->address = pc_value;
   breakpoint_remove(target, breakpoint_p->address);
   breakpoint_p->address = (pc_value==SWI_ADDR) ? SWI_ADDR+4 : SWI_ADDR;

6  if ( !contextid && !breakpoint_p->asid ) {
     breakpoint_add(target, breakpoint_p->address, BKPT_LENGTH, BKPT_HARD);
   } else if( contextid ) {
     breakpoint_p->asid = contextid;
     hybrid_breakpoint_add(target, breakpoint_p->address, breakpoint_p->asid, BKPT_LENGTH, BKPT_HARD)
         ;
11 } else {
     arm = target_to_arm(target);
     arm->mrc(target, 15, 0, 1, 13, 0, &contextid);
     breakpoint_p->asid = contextid;
     hybrid_breakpoint_add(target, breakpoint_p->address, breakpoint_p->asid, BKPT_LENGTH, BKPT_HARD)
         ;
16 }
```

Listing 5.10: Implementation of Hardware Breakpoint Management

Furthermore, as OpenST aims to analyze a specific Linux process, the context ID register must be read to insert hybrid breakpoints (trigger on context ID and on address). It is possible to access this register by the co-processor15 CONTEXTIDR register, c13 as ARM documentation indicates(line 12 of the previous snippet). Otherwise, OpenST performs a system-wide analysis.

5.2.2 Phase 2.2: System Call Tapping

OpenOCD has already implemented a callback system which triggers its execution whenever a debug event occurs. OpenST takes advantage of this system and just checks whether the debug event is a breakpoint or not as it is possible too see in the next snippet.

```
     if(event == TARGET_EVENT_HALTED && target->debug_reason == DBG_REASON_BREAKPOINT)
2    {
       //...
```

Listing 5.11: Implementation of Hardware Breakpoint Management

As the hardware breakpoint is only set at the *swi_vector*'s address, whenever the control flow enters this statement, it should be because the target's CPU is processing a system call.

5.2.3 Phase 2.3: Process Data Structure Reconstruction

The way that the kernel stores and manages its processes information is worth documenting in order to provide a general overview to the reader. The main structure that holds all the information about each process is named *task_struct* and it is known as *process descriptor*. This structure is a double-linked list with every process in the operating system. Among other information, this struct contains the process identification, *pid*, the thread group identification, *tgid* and the executable name, *comm*, which is the information that OpenST collects from the processes which issue system call. This information allows the proposed to associate system calls to processes. The way that OpenST collects that information is by reading the memory of the emphtask_struct at specific offsets, which were gotten by the kernel image and are hardcoded in the proposed tool. The following snippet shows how this information is read.

```
    /* mdw task_struct_addr */
    task_struct_value = *((uint32_t*) get_address_value(target, task_struct_addr, WORD_SIZE));

    pid_addr = task_struct_value + PID_OFFSET;
    comm_addr = task_struct_value + COMM_OFFSET;

    /* mdw pid_addr */
    pid_value = *((uint32_t*) get_address_value(target, pid_addr, WORD_SIZE));

    /* mdw tgid_addr */
    tgid_value = *((uint32_t*) get_address_value(target, pid_addr+4, WORD_SIZE));
```

Listing 5.12: Process Data Reconstruction

However, in order to access the *task_struct* at runtime, one should go through the *thread_info* struct first. The latter one is a struct, located at the bottom of the stack[1], that holds the following information:

```
struct thread_info {
  unsigned long flags;
  int preempt_count;
  mm_segment_t addr_limit;
  struct task_struct *task;
  struct exec_domain *exec_domain;
  __u32 cpu;
  __u32 cpu_domain;
  struct cpu_context_save cpu_context;
```

[1]for stacks that grow down

```
  __u32 syscall;
  __u8 used_cp[16];
  unsigned long tp_value;
  struct crunch_state crunchstate;
  union fp_state fpstate __attribute__((aligned(8)));
  union vfp_state vfpstate;
#ifdef CONFIG_ARM_THUMBEE
  unsigned long thumbee_state;
#endif
  struct restart_block restart_block;
};
```

Listing 5.13: thread_info definition [lxr FreeElectrons, 2012b]

As it is possible to see in the previous snippet, OpenST can access the *task_struct* if it has access to the *thread_info*. Fortunately, it is also possible to access *thread_info* just by having the information of the stack pointer(SP), that it is depicted bellow and it is used on the kernel:

```
static inline struct thread_info *current_thread_info(void)
{
  register unsigned long sp asm ("sp");
  return (struct thread_info *)(sp & ~(THREAD_SIZE - 1));
}
```

Listing 5.14: current_thread_info definition [lxr FreeElectrons, 2012a]

The size of the thread is 8192, thus, the previous snippet masks out the least significant 13 bits from the stack pointer(SP), since 8191 in hexadecimal is 0x00001FFF. After having access to the thread_info, the tool is able to access all the needed information as previously explained.

5.2.4 Memory Introspection & Argument Unmarshalling and Logging

OpenST uses an array of function pointers to organize the previously generated code in **Phase 1: Code Generation**(section 5.1). That array declaration can be seen in the following snippet:

```
static char* (*sys_ptr[NUM_SYSCALLS])(int depth, struct target *target);
```

Listing 5.15: Declaration of the array of function pointers

Given that the code is already generated, the only step that it needs to be performed is to populate this array by assigning each system call id to the pointer of the introspection procedure, which is done as follows:

```c
static void insert_dump_functions_references(void)
{
    sys_ptr[66] = &dump_sys_setsid;
    sys_ptr[2] = &dump_sys_fork;
    sys_ptr[120] = &dump_sys_clone;
    sys_ptr[190] = &dump_sys_vfork;
    sys_ptr[11] = &dump_sys_execve;
    sys_ptr[270] = &dump_sys_arm_fadvise64_64;
    // ...
    sys_ptr[365] = &dump_sys_recvmmsg;
    sys_ptr[102] = &dump_sys_socketcall;
}
```

Listing 5.16: Populate the array of function pointers

In the end, the only thing that is still missing is the system calls logging to the user and to a file. The code organization previously explained makes it possible to read the parameters as follows:

```c
if(sys_ptr[syscall_id])
{
  param_str = sys_ptr[syscall_id](depth_level, target);
  LOG_SYSCALL(pid_value, tgid_value, comm_value, syscall_id, param_str);
  if (fp_trace)
    fprintf(fp_trace, "[pid:%d tgid:%d comm:%s] %s(%s)\n", pid_value, tgid_value, comm_value,
        syscalls_map[syscall_id], param_str);
  free(param_str);
}
```

Listing 5.17: System calls logging

The sys_ptr reads the system call parameters accordingly to the depth level passed as parameter[1]. The parameters string is then logged, along with the other information gotten about the process data from the registers, to the user command line and then appended to a file. The LOG_SYSCALL is a macro included on OpenOCD to allow OpenST to print colored logs.

```c
#define LOG_SYSCALL(pid_value, tgid_value, comm_value, syscall_id, expr ...) \
  LOG_INFO("["GREEN"[pid]%d [tgid]%d [comm]%s" \
  DEFAULT"] "RED"%s"DEFAULT"(%s)", \
  pid_value, tgid_value, comm_value, \
```

[1]depth level can be set before or during analysis

```
syscalls_map[syscall_id], expr)
```

Listing 5.18: LOG_SYSCALL macro definition

Chapter 6

System Details and Architecture

6.1 System Environment Architecture

This section presents the final architecture of OpenST and further information about its protocols and components. Moreover, it explains the actions that are automated to have a final working dynamic analysis tool. The final architecture is depicted on figure 6.1

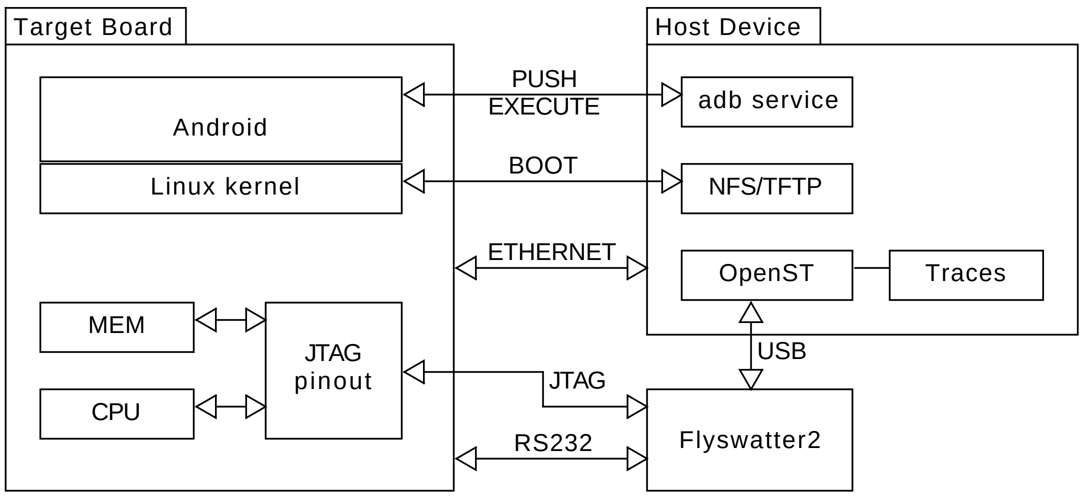


Figure 6.1: System Architecture

6.1.1 Debugging Architecture

The host device has four running processes: OpenOCD, NFS server, TFTP server and adb[1] server.
OpenOCD, as previously mentioned, is the software debugger which enables the possibility to issue JTAG commands to debug the target. These commands are executed with the aid of the JTAG debugger (Flyswatter2) that will send them to the target through JTAG protocol. Afterwards, it will receive the requested information. Flyswatter2 also allows the host device to communicate with the PandaBoard through UART[2]. This feature is useful for booting purposes explained in the next subsection 6.1.2. It is worth noting that it was crucial to debug the configurations employed in the bootloader's (u-boot) configuration. Lastly, adb server is used to install and run Linux programs and Android Apps using the USB OTG[3] PandaBoard's port.
The automation of this debugging process will leverage the capabilities of OpenOCD scripting language and adb shell to automatically initiate the tracing mechanisms, install a malware, and trace its execution. The next figure depicts the debug cycle: Traces are collected using OpenOCD and stored in the host PC. Afterwards there is an offline reconstruction of the Android malware, based on techniques like the ones employed in CopperDroid [K. Tam and Cavallaro, 2015], from the collected traces during runtime.

6.1.2 Booting Schema

One mandatory requirement is to restore the filesystem and the Android image to guarantee a non-infected device before every run. For that purpose, it was needed to create a booting schema. In order to automatically boot the PandaBoard, one needs to have:

- a bootloader that supports network operations;
- a NFS server with the android root filesystem;

[1]Android Debug Bridge
[2]Universal asynchronous receiver/transmitter
[3]USB on-the-go

- a TFTP server that contains Android kernel and the device tree blob of PandaBoard ES

Embedded systems, like Pandaboard ES, have an universal bootloader Engineering [1999] that supports network operations. In order to properly boot Android over Ethernet in PandaBoard, it is used an SD card with a boot partition that contains the uboot image (*uboot.img*), the first level bootloader((*MLO*)) and a bootscript(*boot.scr*). The MLO will boot u-boot that, on its side, will try to execute the boot script file (if existent). The boot script automatizes the booting process, which works as depicted in figure 6.3. In order to boot properly the device, u-boot needs two files: *uImage*; *device tree*. The *uImage* contains the kernel image and the *device tree* file which describes the hardware layout. Furthermore, the kernel needs a filesystem to boot. So, as depicted in figure 6.3, u-boot requests those files over the network (Ethernet connection). The first two files (*uImage* and *device tree*) are located on a TFTP server that is running in the host device. The root filesystem is provided by the NFS server which also runs on the host device. Since the host device is isolated from the environment that runs the linux programs, it will be very unlikely or even impossible that a malware infects the NFS server, the TFTP server, their files or even the host device. The *uImage* and *tree.dtb* are gathered with the *tftp* command of u-boot. Booting the kernel requires

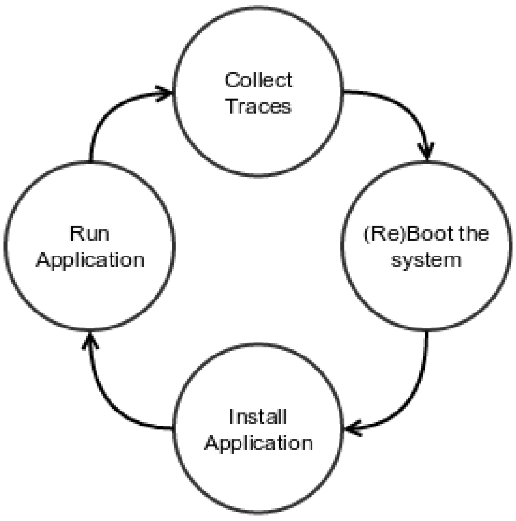


Figure 6.2: Debug Cycle

bootm, which requires bootargs. The bootargs will define booting arguments like the mounting point (devnfs), its type (nfs) and its location (ip address of the nfs server).

6.2 System Requirements

In this section, the reader will get information about the system requirements of every component used for this work. Firstly, there is a brief description containing all the features that the system must have to perform the intended tasks. Lastly, there is a description of the system requirements of each component (the board and the other ones described in the subsection 3.2.3).
The objective of this project is to reconstruct the behavior of Android malicious Apps by analyzing its runtime traces. It is worth noting that the key point of this analysis is that it won't be allowed to used debugging flags nor have access to the malware source code. This way, it will be harder for the malware to un-

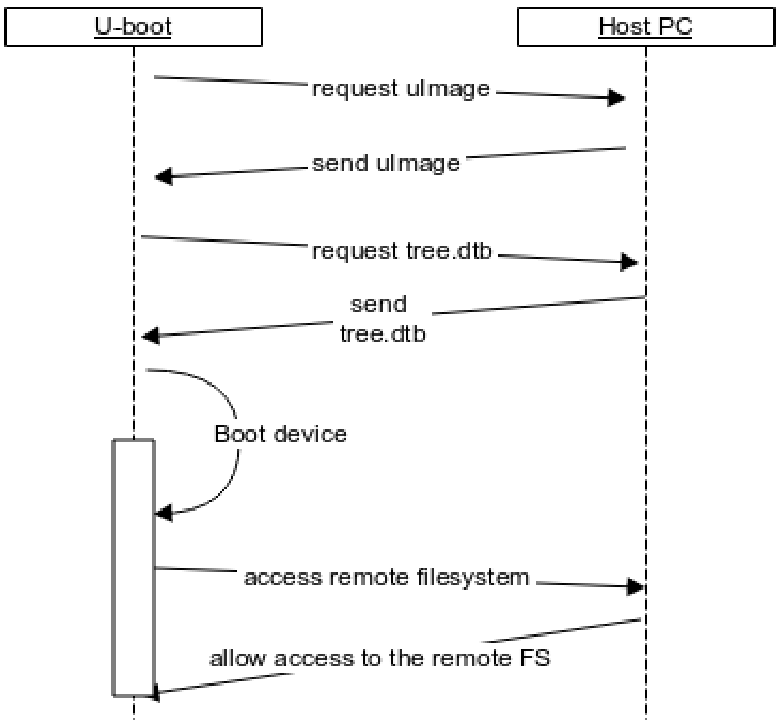


Figure 6.3: Booting Schema

derstand that it is being analyzed. In order to develop an hardware-based debug environment for a given target SoC [1], the environment needs to have:

- a host device;
- a low-cost board based on one SoC that supports Android;
- a JTAG in-circuit debugger/programmer for ARM processors;
- a Software that interfaces with a hardware debugger's JTAG port.

The host device connects to the board through the JTAG debugger, so it can debug the target CPU. Additionally, it should be possible, for the host device, to seamlessly and automatically install Android Apps and Linux programs into the target board, that is running a slightly modified Android OS version. In order to make sure that the system is not infected after each application execution, two features must be added: download the Android OS image from the host and boot the board every run; maintain an unchanged remote filesystem. The host device does not have any special requirement as long as it supports the software used to retrieve traces with some specific JTAG debugger, Ethernet port and two USB ports. On the other hand, the JTAG debugger, the board and the software have some restrictions that should be followed in order to select them.
The following sections contain the analysis of those restrictions to choose wisely the technologies for the environment.

6.2.1 Board

The market offers us many boards. For the sake of simplicity, the reader can see the table **??** with the board's minimum requirements (plus the CPU clock speed). We inserted in the table every board that:

- supports an unmodified or slightly modified AOSP[2] version;
- has an Ethernet port;
- has JTAG pinout;

[1]System on a chip
[2]Android Open Source Project

- has an ARM based CPU;
- supports UART (like RS232)

Supporting AOSP and being an ARM based CPU is mandatory in order to recreate as close as possible a real environment. As previously referred ARM-based processors are the most common ones in the smartphones market [ARM®, 2015]. The support of an Ethernet port is related with the process of booting and maintaining a remote filesystem, which was previously explained. JTAG pinout is also essential since it is the standard that it is used to debug the target SoC. Lastly, the UART port makes possible to have an independent port from the JTAG port to communicate with the target device.
Furthermore, it is of paramount importance to restore the all system between every run, to guarantee the same conditions to every application.

The aforementioned restrictions were not enough to make a straightforward choice - which the reader can check in the table **??**. The processor's clock rate is just displayed to possibly differentiate boards between each other. As we are in the cutting edge, we searched tutorials to connect the board with the JTAG debugger and, in the end, we gave preference to the board which was mostly used in terms of JTAG debugging: Pandaboard ES. The selected board has a TI OMAP4460 OMAP$^{\text{TM}}$processor, which is based on ARM Cortex-A9 architecture.

6.2.2 JTAG Debugger

First of all, between, the JTAG debugger and the hardware tracers, typically, there is a huge gap between their prices. The reason is simple: hardware tracers need to support ETM tracing with TPIU [1], which implies the support of higher clock frequencies. So, as having a low price is indispensable, we narrowed our choice only to cheaper technologies (even in the JTAG debuggers table **??**, which have some costly devices). Note that JTAG debuggers have the prerequisites to this project, however, they perform a slower analysis. Besides the prices (which leaves some options to choose), similarly to the board's selection, we searched tutorials

[1]Trace Port Interface Unit

that in order to know which of the debuggers would best fit with given boards. We concluded that Flyswatter2 is a debugger that has a decent frequency (considering JTAG specification), is cost-effective and it has good references.

6.2.3 Software Debugger

A Software debugger is installed on the host device and is used to communicate with the SoC target through the JTAG debugger. For what matters to this work, the software just needs to support a way of showing the interrupts generated by applications on Android and introspect system calls parameter's. Any JTAG debugger should be able to do that. However, since it is required that the software debugger can be modified, the choice should be biased toward an open source tool. Chameleon Debugger, MULTI IDE, SourcePoint for ARM and IAR embedded workbench are complete expensive tools, but they are not open source. For that reason, OpenOCD[1] is the software debugger used on this project.

[1]Open On-Chip Debugger

Chapter 7

Experimental Validation

7.1 System setup

The figure 7.1 shows the system setup, which will have its components specifications detailed in this section.
The setup environment uses a Raspeberry Pi2, model B with a quad-core ARM® Cortex™-A7 900MHz CPU and 1GB of RAM as the host device**(A)**. The JTAG debugger**(B)** is Flyswatter2 which has clock frequency upto 30MHz. The target**(C)** is a Pandaboard ES rev B3 with a dual-core ARM® Cortex™-A9 MPCore™ with Symmetric Multiprocessing (SMP) at upto 1.2 GHz each. Finally, the setup has a router**(D)** which enables the possibility of accessing the host remotely with a *ssh* client, which is not a requirement of the tool, but it was employed for comfort purposes, since it enables the access to the tool from everywhere without the need to set it up over and over.

7.2 Case Study

OpenST can be used to perform analysis on Linux processes, whether they are Android applications or native Linux programs. This tool tracks the OS-specific behavior by displaying all the contents of the system calls issued by the given Linux process. This tool can be used by someone which wants to understand the behavior of a Linux process.

In this example, a given analyst wants to analyze the system calls issued by an calculator and he uses OpenST to perform the analysis. The analyst connects to OpenST over telnet(localhost on port 4444) and type the following:

```
> systrace bench /path/to/program/executable_name
```

Listing 7.1: Connecting to Open SiliTracer

This command seamlessly installs and executes the binary with adb and then traces the behavior of executable_name. The output is shown on the terminal and it is sent to a file named trace_executable_name.
As it analyzes the OS-specific behavior of a Linux process, it can analyze any native program or Android application. This tool can be used to understand the system calls issued by a Linux process or to analyze the OS-specific behavior of a malware.

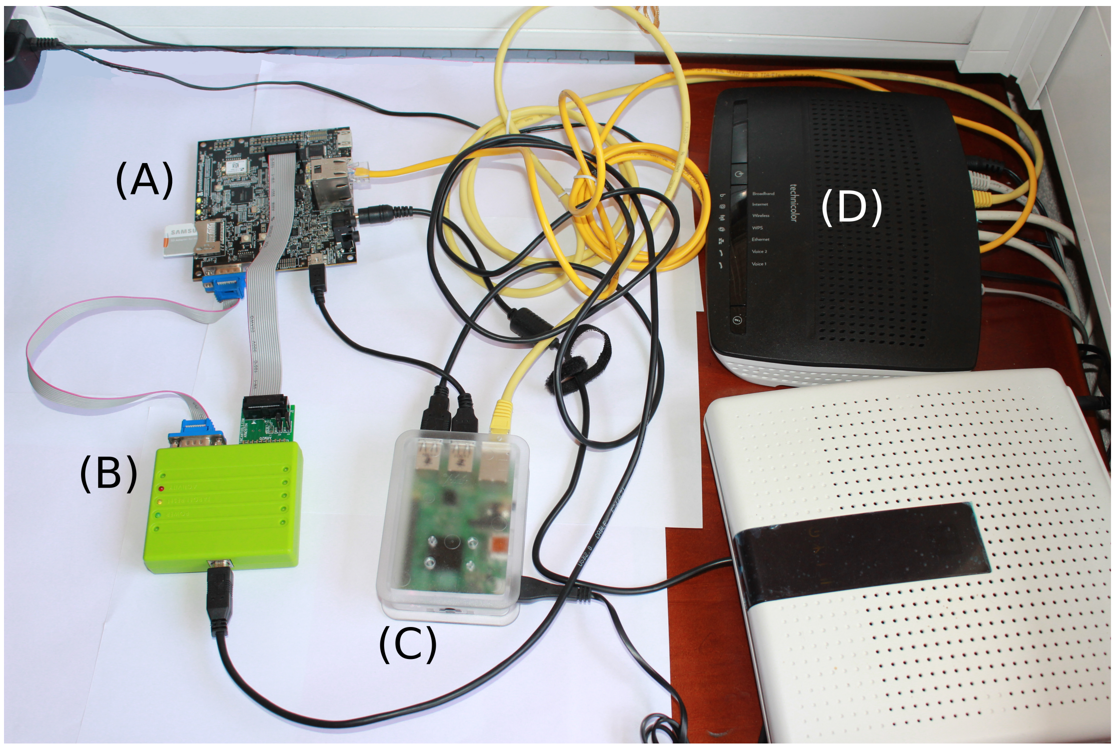


Figure 7.1: System Setup

7.3 Correctness

This section presents the correctness of the results of OpenST. The correctness validates the tool on its output showing whether OpenST tracks all the system calls or it fails some of them. In order to check that, I made an Assembly program for ARM which receives the system call number by parameter and issues the system call with all the arguments set to 0. Then, with another program made in C, a given set of system calls was issued. The output is in the listing **??** of the Appendices. It is possible to note that there are other system calls there due to printing functions, the exit of the program and the starting of the program. The set of the system calls used to test the correctness was the following:

- 5 - sys_open;
- 6 - sys_close;
- 10 - sys_unlink;
- 11 - sys_execve_wrapper;
- 20 - sys_getpid;
- 26 - sys_ptrace;
- 39 - sys_mkdir;
- 40 - sys_rmdir;
- 41 - sys_dup;
- 54 - sys_ioctl;
- 63 - sys_dup2;
- 125 - sys_mprotect;
- 146 - sys_writev;
- 199 - sys_getuid;
- 201 - sys_geteuid;
- 202 - sys_getegid;

- 210 - sys_setresgid;
- 212 - sys_chown;
- 213 - sys_setuid;
- 214 - sys_setgid;
- 217 - sys_getdents64;
- 281 - sys_socket;
- 283 - ABI(sys_connect, sys_oabi_connect)

As it is possible to see from the result, OpenST traces every system call perfectly.

7.4 Benchmarking

This section describes the benchmarking employed to measure the performance hit of OpenST. To summarize, there are two levels of performance hit: micro and macro. In the micro level(subsection 7.4.1), the finest level of OpenST is tested - system call. On the other hand, the macro level(subsection 7.4.2) is the benchmarking of well known programs.

7.4.1 Micro Benchmark

Micro benchmarking is meant to measure OpenST the performance hit on every system call. In order to accurately measure the time that each system call takes to execute on the target, one cannot use the system call clock_gettime since it introduces the system call overhead of the system call itself. Therefore, I had to perform the measurement based on the CPU ticks count from the processor, that is multiplied for the maximum CPU clock frequency. As I had a JTAG debugger, enabling the access to the CPU is straightforward[1]:

```
> halt
> arm mcr 15 0 9 14 0 1
> resume
```

Listing 7.2: Enable the CPU ticks count register

[1] Otherwise it would be needed to create a kernel module

Then in the assembly code, it is only needed to reset the ticks counter and then read the CPU ticks before and after the system call, which achieved as follows:

```
mov r11, 0x17
mcr 15, 0, r11, c9, c12, 0 @reset the counter
mrc 15, 0, r8, C9, C13, 0 @read the CPU ticks
svc 0    @issue syscall
mrc 15, 0, r9, C9, C13, 0
```

Listing 7.3: Read the CPU ticks count register

Furthermore, it was tested the performance impact of the JTAG speed, running the analysis with three levels of frequency speed: 290kHz, 2900kHz and 29Mhz, which are shown in table 7.1. The results of that table clearly show that OpenST generates a great overhead on each system_call. Moreover, it seems that, even if the JTAG clock frequency would be greater (with another JTAG debugger), the performance hit would not be much lower, since the gain is low switching from 2900Khz to 29Mhz. Therefore, I presume that the bottleneck of this approach is the latency of the communication channel.

The micro benchmarking performed is measured in three figures 7.2. The first one(Micro Bench Native) depicts the time spent for each system call on the target. The second figure(Micro Bench with Instrumentation) shows the instrumentation performance hit. The timing related with the instrumentation is almost the same because the operations performed on OpenST are almost the same on each system call. Lastly, the figure(Micro Bench Slowdown) presents the overall time spent on each system call.

Table 7.1: Micro benchmarking results

	Execution time(ms)	
	Average Time	**Standard Deviation**
Native	7.46e-04	5.78e-04
OpenST@29Mhz	178	21
OpenST@2900Khz	201	30
OpenST@290Khz	402	20
	Slowdown@29Mhz	
	4.41e+05	2.99e+05

7.4.2 Macro Benchmark

The macro benchmark, as stated before, aims to compare the execution time of a widely known programs(7zip, ps, nestat) natively and with the analysis of OpenST. The table 7.2 shows the results of that benchmark. The results of this benchmark are better, in the sense that, overall, the performance hit is lower. Such behavior is expected since the programs have many instructions that do not issue system calls (as opposed to the one used on micro benchmark).

7.5 Conclusions

OpenST is able accurately trace all the system calls issued by a specific Linux process, which makes it possible to understand the OS-specific behavior of it. This works for every native program or Android application. However, the performance hit of the analysis is significant and the hardware setup should be optimized. Nevertheless, the tool fulfills its purpose of porting accurately a VMI-based technique to Hardware, raising the bar for malware writers to evade this dynamic analysis tool.

Table 7.2: Macro benchmarking results

		Execution time(s)		**Slowdown**
		Average Time	**Standard Deviation**	
7za	native	2.55	1.3	69.1 ± 140
	OpenST	176	31	
ps	native	0.699	0.56	243 ± 833
	OpenST	121	25	
netstat	native	0.0192	0.0035	635 ± 253
	OpenST	116	31.4	

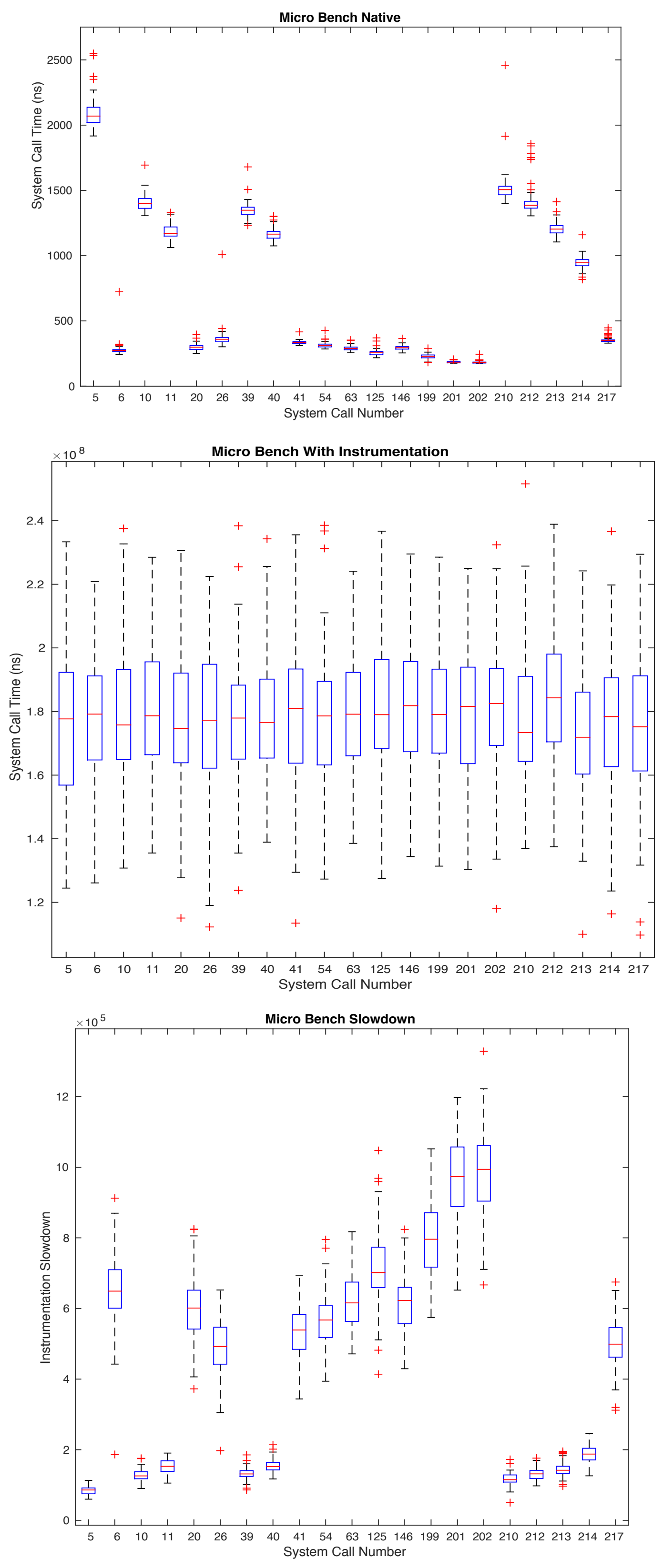


Figure 7.2: Micro benchmarking on the target

Chapter 8

Discussion

8.1 Limitations

This section provides the limitations of OpenST and some discussion around them. The main limitation of OpenST is related to the performance of the analysis. As previously analyzed in table 7.1, the speed of the clock frequency of the JTAG debugger influences the overall performance of the analysis. Even though, the maximum frequency of the Flyswatter2 is 30Mhz, it seems that even if it would support the maximum speed of JTAG protocol(100MHz), the performance improvement would not be significant. Thus, I presume that the performance hit is related with the latency of the whole communication. Additionally, there is a performance hit that depends on the target's scheduler. The analysis starts by breakpointing just the address of *swi_vector* and, thus, performs a system-wide analysis until it finds the process context ID to set a hybrid breakpoint(address and context ID).

The lack of support of SMP on Pandabord ES on OpenOCD is also a limitation of this approach. As referred before, this the only change made on the kernel so that it is ensured that it runs on a single CPU.

OpenST, even though it can record information on Android Applications, do not get specific information on Android behavior. So, a security expert just can evaluate an application considering its OS-specific behavior.

Overall, as existing approaches it does not support real-time. The comparison of the performance of OpenST with VMI-based approaches is not straightforward, since the benchmarks provided by the literature do not take into consideration the overhead of the emulator. Moreover, it is still possible to evade our analysis. As referred in one of the analyzed studies [T. Vidas, 2014], the analysis of our system will have a fixed timespan to trace the runtime events of each malware sample, so it is possible to evade our system by starting the malware's malicious activities only after that defined timespan. It is worth mentioning though, that even in that case, the malware is not distinguishing our environment from a another which is not being analyzed.

Chapter 9

Conclusions and Future Work

This work achieved its purpose on performing dynamic analysis on Linux processes with the aid of the created tool, Open SiliTracer. This automated tool uses hardware memory introspection to perform dynamic analysis on Linux processes. It is the very first effort on this kind of analysis using open source solutions, because it does not rely on hypervisors. However, this tool needs some future work regarding the limitations presented in the previous chapter.

9.1 Future Work

The novelty of the approach is implemented on OpenST, however, for the tool to move to the next stage it should record not only OS-specific but also Android specific information so that a security expert can use it for tracing Android applications. It can do it so if it further inspects the ioctl system call. The reason for not introducing this on this work is two-fold:

- It is already implemented on the literature [K. Tam and Cavallaro, 2015] so it does not constitute novelty;
- It is a matter of coding; there is nothing new in our approach

In order to increase overall performance other system setups(target, host, dongle) should be employed. Finally, OpenST needs to be tested with multicore and against existing fingerprinting techniques.

References

(2007). *Proc. of the Linux Symposium(SYMP 2007)*, volume 2. 46

Android-central (2015). Daily android activations. 1

ARM® (2010). Arm virtualization extensions. 23

ARM® (2012). Coresight components. x, 28

ARM® (2015). Arm popularity. 2, 26, 66

ARM®InformationCenter (2011). Embedded trace macrocell™. 29

Bellard, F. (2005). Qemu, a fast and portable dynamic translator. In *USENIX ATC*. 5, 7, 11, 17

Bramley, J. (2010). Caches and self-modifying code. 22

DEXLabs (2012). Dexter. 2

eliben (2015). pycparser. 46, 50

Engineering, D. S. (1999). U-boot. 63

F-SecureLabs (2014). Mobile threat report. 1

Garfinkel, T. and Rosenblum, M. (2003). A virtual machine introspection based architecture for intrusion detection. In *NDSS*. 2, 4, 7

Google (2012). Google bouncer. 2

IDC (2014). Smartphone os market share. 1

InternationalSecureSystemsLab (2012). Anubis. 2, 5

K. Tam, S. J. Khan, A. F. and Cavallaro, L. (2015). Copperdroid: Automatic reconstruction of android malware behaviors. In *Network and Distributed System Security (NDSS) Symposium*, San Diego, CA, USA. x, 2, 4, 5, 14, 39, 62, 77

lxr FreeElectrons (2012a). Get current thread info. xii, 58

lxr FreeElectrons (2012b). Thread info struct. xii, 58

Matenaar, F. and Schulz, P. (2012). Detecting android sandboxes. 17, 22

McAfee (2014). Mcafee labs threats report. 1

Oberheide and Miller (2012). Dissecting the android bouncer. 2

OpenIntents (2007). Sensorsimulator. 21, 23

R. Paleari, L. Martignoni, G. F. R. and Bruschi, D. (2009). A fistful of red-pills: How to automatically generate procedures to detect cpu emulators. In *WOOT*. 15

Rath, D. (2005). Open on-chip-debugger. 26, 45, 55

Rutkowska, J. (2006). Blue pill. 15

Symantec (2012). Norton cybercrime report. 1

T. Petsas, G. Voyatzis, E. A. M. P. and Ioannidis, S. (2014). Rage against the virtual machine: Hindering dynamic analysis of android malware. In *EuroSec*. 2, 4, 20, 21

T. Raffetseder, C. K. and Kirda, E. (2007). Detecting system emulators. In Springer, editor, *international conference on Information Security*, 10, pages 1–18. 17

T. Vidas, D. V. and Christin, N. (2011). All your droid are belong to us: A survey of current android attacks. In *WOOT*. 17

T. Vidas, N. C. (2014). Evading android runtime analysis via sandbox detection. In *ASIA CCS*. x, 2, 4, 16, 18, 20, 76

the honeynet project (2012). Droidbox. 2, 5

trend micro (2014). The mobile cybercriminal underground market in china. 1

V. Rastogi, Y. C. and Jiang., X. (2013). evaluating android anti-malware against transformation attacks. In *ASIA CCS*. 2

VRT (2013). Changing the imei, provider, model, and phone number in the android emulator. 20

W. Zhou, Y. Zhou, X. J. and Ning, P. (2012). Detecting repackaged smartphone applications in third-party android marketplaces. In *CODASPY*. 1

Yan, L.-K. and Yin, H. (2012). Droidscope: Seamlessly reconstructing os and dalvik semantic views for dynamic android malware analysis. In *USENIX Security*. x, 2, 4, 5, 8, 11, 39