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

OpenST: Feasibility Study and Prototype of a Low-cost, Hardware-based System Call Tracer

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

Tesi di laurea di:
Chengyu Zheng Matricola n. 820324

Anno Accademico 2014-2015
Facoltà di Ingegneria dell' Informazione

Sommario

C'è necessità di nuovi approcci che permettano agli esperti di sicurezza di analizzare e capire il comportamento di malware, o programmi sconosciuti, sul sistema analizzato. In letteratura sono state proposte sandbox hardware per analisi di malware per sostituire quelle basate su emulazione, per via della loro maggiore trasparenza. Una delle caratteristiche fondamentali di una sandbox è la sua capacità di tracciare le operazioni che compie sul sistema (e.g., istruzioni macchina, chiamate di sistema). Nello stato dell'arte, le sandbox basate sull'emulazione utilizzano tecniche di virtual machine introspection (VMI), che consistono nel tracciare le istruzioni da fuori la macchina virtuale per ricostruire eventi di alto livello come chiamate di sistema. Il tracciamento su sandbox basate su hardware è ancora un problema aperto, in quanto è fortemente dipendente dalla capacità di debug della CPU. È interessante notare che, la maggior parte dei dispositivi mobili (che sono tra gli obiettivi di autori di malware) sono basati su architettura ARM e quindi supportano nativamente il debugging a livello macchina.

In questo lavoro studiamo la fattibilità di implementare un tracer delle chiamate di sistema per Android/Linux in esecuzione su processori ARM. OpenST propone uno strumento open source che sfrutta l'interfaccia JTAG per implementare l'equivalente di VMI in hardware. Più precisamente, il nostro strumento utilizza breakpoint hardware per monitorare i software interrupt (istruzione SWI) e leggere i registri della CPU per la loro ricostruzione. OpenST ispeziona anche il processo in esecuzione di memoria per ricostruire il valore degli argomenti passati alla funzione di sistema, ed eseguendo dereferenziazione dei puntatori e unmarshalling dei dati in base alle esigenze. OpenST è portabile su differenti versioni di Linux perché ricostruisce i prototipi delle chiamate di sistema dall'immagine binaria del kernel, da cui generiamo automaticamente le procedure automatiche per l'unmarshalling.

Abbiamo implementato OpenST e valutato la sua correttezza con un'applicazione di test che invoca alcune chiamate di sistema. Inoltre, abbiamo effettuato micro e macrobenchmark su 3 applicazioni di uso comune. I risultati del micro-benchmark mostrano che la necessitá di mettere in pausa la CPU per leggere la memoria per ricostruire i valori di argomenti impone un overhead significativo, intorno a 180 ms, laddove una chiamata di sistema utilizza circa 500– 2000ns. All'attuale stato dell'arte le sandbox basate su emulazione impongono un overhead di una frazione di millisecondo. I nostri macro- benchmark dimostrano che questo overhead ha un impatto di 70x, in media, il tempo complessivo di esecuzione. In pratica, i nostri test con applicazioni Android hanno dimostrato che questo rallentamento rende l'interfaccia utente inutilizzabile. Abbiamo misurato che il overhead dipende dalla velocità della scheda JTAG, quindi, in linea di principio, può essere ridotto utilizzando hardware più veloce. In conclusione, riteniamo che il nostro approccio sia promettente, ma irrealizzabile con l'attuale hardware a basso costo, che è un requisito per un utilizzo massivo nell'analisi di malware.

Abstract

There is a need for appropriate analysis approaches that allow researchers to understand what malware, or generic unknown programs, do on the target system. Hardware-based malware-analysis sandboxes have been recently proposed to replace emulator-based sandboxes, thanks to their transparency and resilience to emulator-detection attacks. A core part of any sandbox is its capability of "tracing" a (malicious) running program, such that the actions (e.g., instructions, operating system calls) that it performs on the system can be observed. In state-of-the-art emulator-based sandboxes tracing relies on so-called virtual machine introspection (VMI) techniques, which consist in tracing the instructions from outside the virtual CPU for reconstructing high-level events such as system calls. In hardware-based sandboxes tracing is still an open problem, as it is highly dependent from the debugging capability of the CPU. Interestingly, we observe that the vast majority of mobile devices (which are among the targets of malware authors) are based on the ARM architecture, which natively supports machine-level debugging from hardware interfaces.

In this work we assess the feasibility of implementing a system call tracer for the Android/Linux operating system running on ARM-based computers. We propose OpenST, an open-source tool that leverages the JTAG interface to perform the equivalent of VMI yet in hardware. More precisely, our tool uses hardware breakpoints to track the occurrence of software interrupts and inspect the CPU registers in order to reconstruct system calls. OpenST also inspects the running process' memory to reconstruct the value of each argument passed to the system function, performing pointer de-referencing and data unmarshalling as needed. OpenST is portable across Linux versions because it derives the system call prototypes from the kernel binary image, from which it generates argument-unmarshalling procedures automatically.

We implemented OpenST and evaluated its correctness against a testing Linux application that invokes known system calls. Moreover, we performed micro- and macro-benchmarks on 3 real-world applications. Our micro-benchmarks show that the need for pausing and resuming the CPU to inspect the memory for reconstructing the arguments values imposes a substantial overhead, around 180ms, where a system call takes 500–2000ns on average. In comparison the state-of-the-art emulator based sandbox imposes an overhead of a fraction of the millisecond. Our macro-benchmarks show that this overhead has an impact of 70x on average on the overall execution time. In practice, our tests with Android applications showed that this slowdown makes the user interface unusable. We measured that the overhead depends from the speed of the JTAG adapter, so, in principle, it could be reduced by using faster hardware. In conclusion, I believe that our approach is promising yet unfeasible with current low-cost hardware, which is a requirement for large-scale malware analysis.

Acknowledgements

On completion of my Master thesis I would like to sincerely thank all those who supported me in realizing and finishing my work.

First of all, I am heartily thankful to my supervisors spending time and effort on me. Thoughout all of the stages of my thesis, they helped me to help me to focus the research on the right direction, seriously revised all of my work, and patiently help me resolve issues when I am stuck at some point. Additionally, I really appreciate their will for hosting and motivating me all the time while working at NECSTLab at Politecnico Di Milano. When meeting Prof. Federico Maggi and Stefano Zanero on Computer Security course at School, I was really impressed by their will to put real world application into the focus of the class. This honest approach toward research interaction and progress allowed me to get to know several interesting and, more important, very kind researchers giving me valuable input for current and future research problems and directions. Moreover thankful given his impressive expertise and experience in malware and security research were a valuable source for my work.

Furthermore, I want to thank the Politecnico Di Milano for funding the NECSTLab. While working in the lab I learned a lot from other people. The people working are amazing. The time spent here were amazing. I specially appreciated the time spent eating together at lunch and enjoyed discussed related to both technical and non technical issue.

Thank you for giving me so much support.

Contents

Chapter 1

Introduction

In recent years the popularity of the smart-phones has increased exponentially. With an estimated market share of 84%, Android is now the most popular operating system for smart-phones and tablets [1]. With over 500 million Android devices and 50 billion application downloaded, due to their popularity these devices have attracted the cyber-criminals' interests. With an estimated number of malicious applications ranging from 120,000 to 718,000 [2, 3], the research community in computer security and the industry have both recognized the alarming level of threat against mobile devices. Due to the sensitiveness of the data contained in the mobile phones, infecting mobile devices turned out to be a very lucrative, illicit business for malware writers and distributors. For example, in 2012 the Eurograbber malware alone stole more than 36 million euro from some 30,000 retail and corporate accounts in Europe [4]!

With 20,000 new application being released every month it requires malware researchers and app store administrators a reliable and scalable solution for quickly analyzing new apps to identify and isolate malicious applications. The famous Google Bouncer, an automated tool that checks apps submitted to the Google Play Store [5], was shown to be easy to evade [3, 6]. The reason was that the Google Bouncer was based on a device emulator, which could be detected by a malware by exploiting the discrepancies between the real device and the emulated device (i.e., emulators are not perfect). So new method were proposed to address this issue but none of them provide a comprehensive solution to obtain a thorough understanding of unknown applications: Thomas Blasing and Albayrak [7], Alessandro Reina and Cavallaro [8], William Enck and Sheth [9], Vaibhav Rastogi and Enck [10], Michael Spreitzenbarth and Hoffmann [11], and Yan and Yin [12] were all based on using an emulator. Clearly, if an emulator is detected, the malware could refuse to run further and, possibly, exhibit a perfectly benign behavior.

We believe that this issue should be eradicated once and for all. Therefore, motivated by the lack of alternatives, I propose to investigate the feasibility of porting the state-of-the-art approaches to work directly on hardware. While at a first glance this may appear easy, it actually entails several challenges. In essence, the goal of a malware-analysis sandbox is to observe a running process and record interesting events. For this, the best observation point is the user- to-kernel interface, which can be instrumented to record the stream of system calls. In this way, high-level "behaviors" such as "sending a spam email," or "opening a reverse shell on port X," and so forth, can be reconstructed. To this end, the main challenge is the need for introspection into the machine state, in order to observe events such as instructions, interrupts, and registers, that are required to reconstruct the occurrence of a system call. In software emulators, this is easily done by instrumenting the functions that implement the virtual CPU. Similar challenges hold for memory introspection. These first two challenges could be in principle tackled by relying on the debugging capabilities of modern systems on chip development boards, which expose interfaces that can be used to perform the equivalent of emulator instrumentation. This is however not easy, too. On the one hand, so-called trace ports could be leveraged to read the stream of instructions being executed at a high speed, without imposing any overhead as they are embedded in the chip. Unfortunately, to trace system calls it is necessary to reverse engineer, at run time, the semantic of the register values in order to know which system function is executed, and which values in memory are being passed to the function. Therefore, memory becomes stale immediately after an instruction is logged by the trace port. On the other hand, classic, JTAG-attached hardware debuggers could be leveraged to overcome this limitation, by pausing the CPU whenever a system call interrupt is detected, reading up-to-date content from registers and memory, and resuming the CPU. Clearly, this imposes substantial overheads. However, hardware debuggers are relatively inexpensive and modern ARM-based boards (featuring the very same cores shipped with mobile devices) expose interfaces that allow easy access to the information required to perform system call tracing as done in software emulators. In malware analysis, cost is crucial: Software emulators scale much faster and at a lower cost, because they can be deployed and consolidated in cost-effective servers, whereas hardware- based malware analysis sandboxes require a dedicated, physical board per worker. Last, there are minor technical challenges that need to be considered regarding the ease of resetting the sandbox to a clean state after each analysis: This is easily done in a software emulator by restoring the (software) block device and memory snapshot, whereas in hardware-based sandboxes this would require a reboot and restoring the block devices' original content.

Having considered the aforementioned challenges, in this work, I assess the feasibility of porting virtual-machine introspection (VMI) techniques, which are those used in state-of-the-art sanboxes (e.g., [12]), to leverage On-Chip Debuggers attached to low- cost development boards. I focus specifically on assessing the overhead imposed by system call tracing and on the technical feasibility of the overall idea, deferring the development of a refined prototype to future work. In practice, I propose a set of open-source tools and a working prototype, that I name OpenST, to run an executable Linux/Android application and trace the system calls and arguments resulting by the respective process. OpenST is based on automatically generating stub code for each system call that need to be traced. Such stub code is executed on an external computer, connected to the target board through a JTAG adapter, managed by OpenOCD (an open-source On-Chip Debugging framework). Leveraging hardware breakpoints, every time a software interrupt is trapped, I use OpenOCD to pause the CPU, inspect its registers and the target process' memory, and resume it. I generate the stub code automatically, offline, by parsing the system call prototypes from the kernel binary (in DWARF format). More precisely, for each system call definition I parse the data structures of its input arguments and convert them in offsets, which I use to instruct the stub code to unmarshall the argument values from the main memory given a base address.

I implemented OpenST and evaluated it on a testbed Linux application that I developed, which invokes known system calls in a given order. The results show that OpenST correctly intercepts the system calls and their respective arguments.

I performed micro- and macro-benchmarks on 3 real-world Linux applications (`7zip`, `ps`, `netstat`). The results show that the need for pausing and resuming the CPU to inspect the memory for reconstructing the arguments values imposes a substantial overhead, around 180ms, where a system call takes a fraction of this time, 500–2000ns on average. In comparison state-of- the-art emulator-based sandboxes impose an overhead of a fraction of the millisecond. The results of the macro-benchmarks are mildly better, showing that such overhead has an impact of 70x on average on the overall execution time. This would be acceptable in a malware- analysis setting, where an analysis normally takes up to 5–10 minutes. However, in practice, our tests with Android applications showed that this slowdown makes the user interface unusable by a user. Again, this would be acceptable in a malware-analysis setting. Interestingly, I measured that the overhead depends from the speed of the JTAG adapter, so, in principle, it could be reduced by using faster hardware. In conclusion, I believe that the approach of porting VMI to hardware is promising yet unfeasible with current low-cost hardware, which is a requirement for large-scale malware analysis.

In summary, this thesis makes the following contributions:

- Analyzed the market of commercial off-the-shelf (COTS) boards with proper On-Chip Debugging capabilities;
- The first to design a malware-analysis sandbox based on VMI fully on hardware;
- Implement, evaluate and release a prototype of our sandbox for future research.

The source code of OpenST is available for download at `https://github.com/necst/openst`.

Chapter 2

State of the Art and Motivation

This chapter introduces the background concepts necessary to state the problem. some of the most common techniques used by security expert to analyze malicious applications and its limitations.

2.1 Dynamic Analysis

Dynamic Analysis is the analysis of the behavior of a running program. Security experts use these techniques to classify software behaviors as malicious or benign. In fact dynamic are not affected by code obfuscation, runtime packing or anti-debugging techniques. The main problem is the code coverage.

Most dynamic analysis tools used today implements functionality that intercept APIs and system calls. Additionally several analysis tools provide the functionality in order track which sensitive data has been compromised. Automated dynamic analysis tools often create a report about the observed sample. These reports are used to group malware with same pattern. Usually samples that has new pattern are examined manually.

2.2 Virtual Machine Introspection

Virtual Machine Introspection (VMI) [13, 14, 15, 16, 17] is a mechanism that allows indirect inspection and manipulation of the state of virtual machine. The indirection of this approach offers attractive isolation properties that have resulted in a variety of VMI-based applications dealing with security, performance, and debugging in virtual machine environments. VMI allows visibility into and control of the state of a running virtual machine by software running outside of the virtual machine.

In this section I analyze two tools for understanding the behavior of an Android application, which are DroidScope and CopperDroid. VMI is used in both these tools in order to obtain OS-level view. This is done by instrumenting the hypervisor.

OS-level view is described by information about the current process and its system calls.

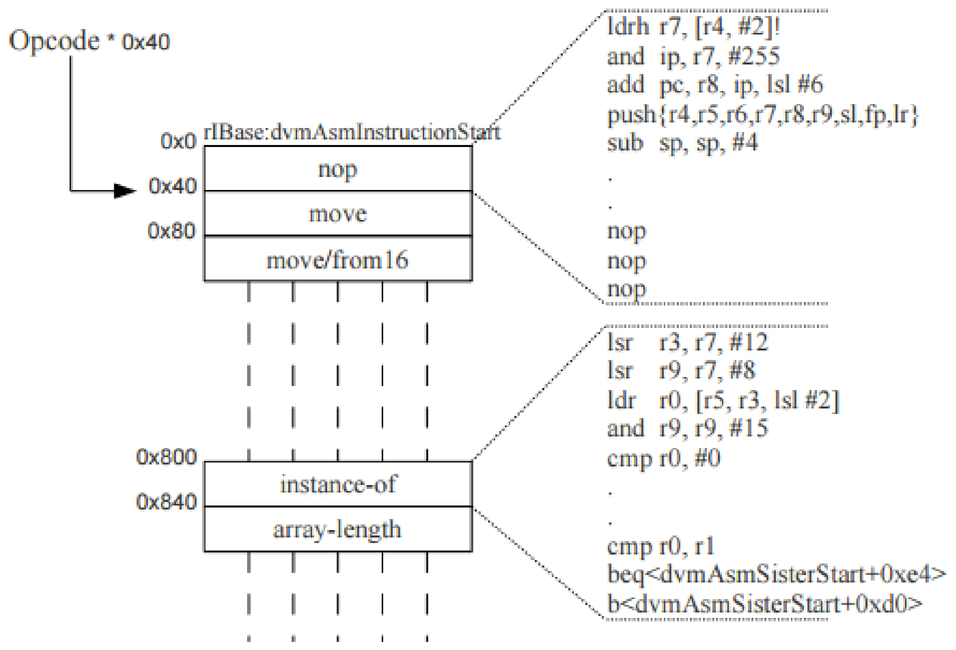


FIGURE 2.1: Dalvik Opcode Emulation Layout. (Left) Dalvik machine Opcode. (Right) host machine code which is translated into

Information about the current process in execution can be obtained by introspecting task_struct of the current process, which can be easily located according to the design of the Linux kernel. The current thread_info structure is always located at the beginning of the stack which can be calculated by ignoring some of the most significant bits (stack pointer & 0x1FFF). The struct thread_info has a pointer which references the current task_struct, that makes it possible to obtain information such as process identification (PID), task group identification (TGID) and the executable name (COMM). This information allows to keep track of specific processes by posing a filter on specific PID or COMM.

A user-level process has to make system calls to access various system resources and thus obtaining its system call behavior is essential for understanding malicious applications. On the ARM architecture, the SWI instruction is used to make system calls with the system call number in register R7. This is similar to x86 where the int 0x80 instruction is used to transition into privileged mode and the system call number is passed through the eax register. To obtain the system call information, SWI emulation code is instrumented by inserting a call callback function that retrieves additional information from memory. For important system calls (e.g., open, close, read, write, connect, etc.), the system call parameters and return values are retrieved as well. As a result, it is possible to understand how a user-level process accesses the file system and the network, communicates with another process, and so on.

DroidScope [12] In addition to the OS-level view reconstruction, it reconstruct the Dalvik semantic, this is done by analyzing the interpreter. The interpreter is part of the DVM and its main task is execute Dalvik bytecode by translating them into corresponding executable machine code.

The interpreter, named mterp, uses an offset addressing method to map Dalvik opcodes to machine code blocks as shown in Figure 2.1. Each opcode has 64 bytes of memory to store the corresponding emulation code, and any emulation code that does not fit within the 64 bytes use an overflow area, dvmAsmSisterStart, (see instance-of in Figure 2.1 3). This design simplifies the emulation of Dalvik instructions. Mterp simply calculates the offset, opcode * 64, and jumps to the corresponding emulation block.

This design also simplifies the reverse conversion from native to Dalvik instructions as well: when the program counter (R15) points to any of these code regions, we are sure that the DVM is interpreting a bytecode instruction. Furthermore, it is trivial to determine the opcode of the currently executing Dalvik instruction. In DroidScope we first identify the virtual address of rIBase, the beginning of the emulation code region, and then calculate the opcode using the formula (R15 - rIBase)/64. rIBase is dynamically calculated as the virtual address of libdvm.so (obtained from the shadow memory map in the OS- level view) plus the offset of dvmAsmInstructionStart (a debug symbol).

CopperDroid [18] CopperDroid in addition to the OS-level view reconstruction, it uses an oracle-based technique to automatically reconstruct Android-specific behavior.

Android-specific behavior (e.g., send SMSs, make calls) is retrieved by analyzing the Inter-Process Communication (IPC). Two processes cannot share memory and communicate with each other directly. So to communicate, objects has to be marshalled in order to communicate across process. This marshalling handles by Android with AIDL (Android Interface Definition Language).

An AIDL file defines a given interface detailing its methods, parameters and return values types. The Android platform includes an AIDL parser which, given an AIDL file, will produce Proxy and Stub classes. The Proxy is used on the client side and matches the method calls that a client would call (in terms of method name, parameters and return value). The Stub, used on the server side, utilizes the transaction code in order to perform the appropriate actions for the given method call. The reason for this is that all Binder calls go through the Binder device driver as I/O controls and while the functions on the client side (Proxy) match those in the client, the server (Stub) needs to efficiently map a given call to its method. The actual parcel data is held in the buffer field of the binder_transaction_data structure (see Figure 2.2).

While the AIDL process works fine for marshalling data between clients and servers during normal runtime, it is necessary for CopperDroid to combine components of the Proxy and Stub in order to unmarshall the objects post-analysis. Furthermore, it is also necessary to implement a parcel reader that understands how to unmarshall parameters from the marshalled data. Therefore, CopperDroid includes a modified AIDL parser that obtains the method names, parameters and return values types (i.e., usually utilized in the Proxy at runtime) to build a mapping between transaction codes and methods. It then combines this information with the parcel reader class mentioned earlier to automatically produce handling code for a given method. CopperDroid utilizes this code to extract the necessary information from any Binder call during later analysis. All this automatically-generated AIDL information is stored in multiple Python files, preserving the mapping of all interface names to parcel data extraction routines. For example, “com.android.internal.telephone.ISms” is mapped to the db_parcel_ISms function call. As this process is only needed once per Android OS version, it can be done before analysis and does not induce overhead during analyses.

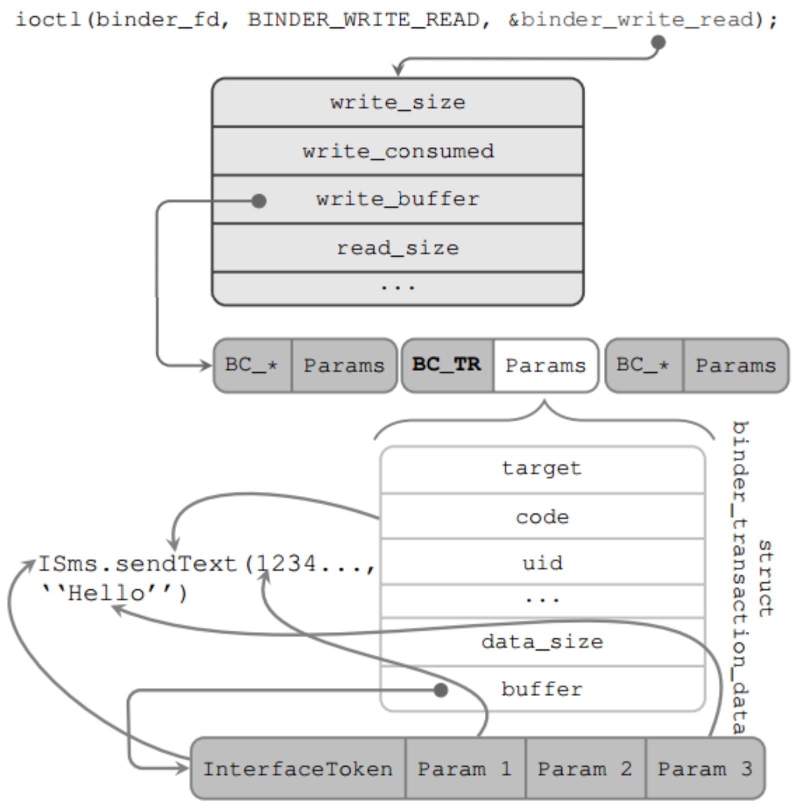


FIGURE 2.2: An example Binder payload corresponding to a send SMS action. CopperDroid initially parses the payload of the ioctl system call (Binder interaction), and sends the extracted (potentially- marshalled) arguments to the unmarshalling Oracle, which automatically unmarshalls them to reconstruct the send SMS behavior of the action under analysis

2.3 On-Chip Debugging

In this section basics information related to the debugging architecture is explained. Specifically the Section 2.3.1 talks about the overall architecture and its component, Section 2.3.2 talks about the hardware debugger and its feature.

2.3.1 ARM Debugging

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

- Host PC;
- JTAG adapter;
- Target Board

The host PC runs a software debugger such as Open On-Chip Debugger (OpenOCD [19]). 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.

2.3.1.1 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 afterward 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 connection can be made in though two interfaces, the SWD or JTAG interface. Then, through an external hardware tool, for instance RealView, it is possible to communicate and perform operations to the DP. The following Figure 2.3 resumes this process.

2.3.1.2 Hardware Breakpoints

Hardware breakpoints are a type of breakpoints which is integrated into the SoC.

In ARM-based processor gives the possible to set hardware breakpoint. These breakpoints are able to stop the target's execution when the program counter execute an instruction stored at certain address. These hardware breakpoint are implemented in hardware as comparators, which takes two input, the first is usually the program counter and the second is the value we sets. Whenever the program counter reach that specific address it will trigger a signal in order to halt the target's CPU.

In order to set breakpoint in Cortex-A processor, *breakpoint register pair*(BRP) must be set in order to set a breakpoint. Each BRP is composed by one breakpoint register control(BRC) and one breakpoint register value(BRV). The BRV register holds the address or the context ID of used by the comparator, the BCR holds additional options of the breakpoint. Possible options are shown in Figure 2.4.

In OpenST we use a combination of hardware breakpoint in order to trace an address and context ID. Two BVR are used to store both address and a context ID and two BRP must be used and linked in order to combine two breakpoint. The bits [20:22] state the

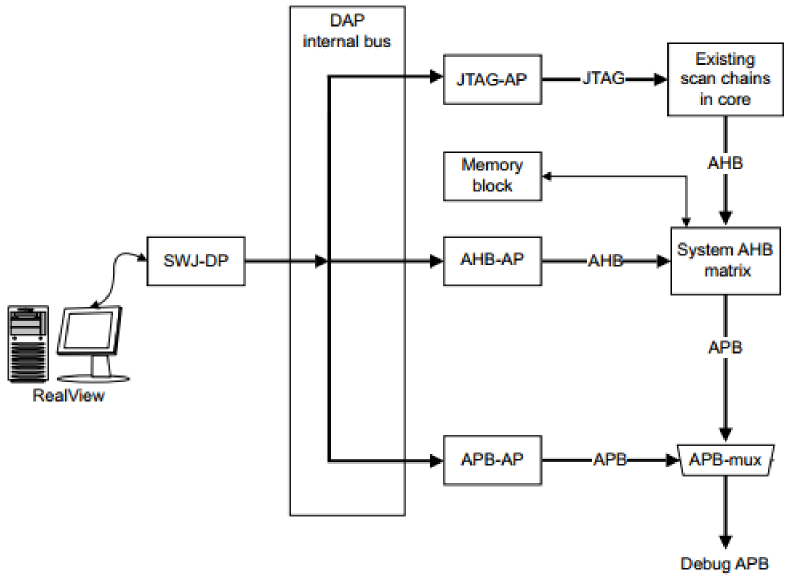


Figure 2.3: The SWJ-DP is a combined JTAG-DP and SW-DP that enables you to connect either a Serial Wire Debug (SWD) or JTAG probe to a target. It enables access either to the JTAG- DP or SW-DP blocks. To make efficient use of package pins, serial wire shares, or overlays, the JTAG pins use an autodetect mechanism that switches between JTAG-DP and SW-DP depending on which probe is connected.

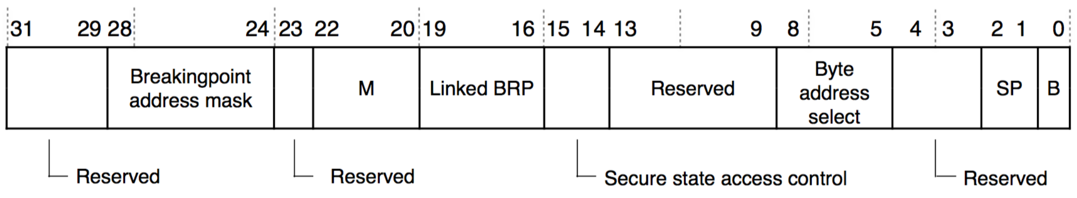


Figure 2.4: Breakpoint control register's bits. Which its possible to use in order to set specific breakpoint.

meaning of the BVR it could be set to either a context ID or a virtual address and bits [16:19] hold the ID of the BRP to be linked so this value should be set in order to have an hybrid breakpoint.

Hardware breakpoints has a good performance but it has a four breakpoints or two hybrid breakpoints. In OpenST only one hybrid hardware breakpoint. Hardware breakpoint are only programmable by using either though JTAG or using privileged co- processor instructions.

2.3.2 JTAG for ARM

In this section I will explain the JTAG and its facilities. Afterwards, there are lists of the JTAG debuggers, hardware tracers and software debuggers available in the market. In order to choose some specific JTAG adapter, it is fundamental a list of features. The selected features are shown in Table 2.1.

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

Legend		
	✓	Mandatory feature
	✗	Not mandatory

Table 2.1: JTAG constraints. Some of the constrains are mandatory in order to proper implement VMI

In the previous table there were a list of mandatory and not mandatory features necessary for the success of this project. For the sake of clarity, features will be explained in the following paragraph. The most important feature are searching JTAG adapter that are compatible with ARM-based processor. Usually they already have incorporated the ability of the inspecting the memory and handling the interrupt. Since most of the devices uses the FT2232C chip produced by FTDI company. Even in that case, in order to reduce any complication during the development of OpenST we sticked with known working solutions. The last two features are not mandatory in to the accomplishment of OpenST. 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 JTAG adapter could have the adaptive clocking (RTCK) which automates the process of maximizing the clock frequency between the adapter and the target board though by negotiating it, so there is no need to set the frequency manually.

2.3.2.1 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. This usually involved replacing the processor temporarily with a hardware emulator. This gives the ability 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.

2.3.2.2 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. The OCD capability is provided by additional silicon within the processor, which adds debugging logics. A downside of this approach is that the debugger might be limited by what feature the manufacture has implemented.

The JTAG standard was designed to assist with device, board, and system testing, diagnosis, and fault isolation. It is an essential way of debugging embedded systems. Today it is used to access the sub- blocks of integrated circuits (ICs). Today it is also used to debug early stage bootloader such as MLO or U-Boot. It allow to debug the wiring of the embedded system through boundary scan testing. Generally, smart-phones have hidden JTAG connector which allow to be interfaced though soldering. JTAG is also widely used for IC debug ports. Embedded systems development relies on debuggers communicating with chips via JTAG to perform operations on CPU and memory, such as breakpointing and firmware flashing.

2.3.2.3 JTAG Debuggers

JTAG adapters are used to access to the target processor's On-Chip Debug modules by using the JTAG protocol. The OCD allow the external adapter to have access to the up to machine instruction level. A comparison of JTAG debuggers available in the market are shown in Table 2.2. Since we have multiple choice that satisfies the constraints, choosing the JTAG debugger was no easy task. Some devices even though in the paper devices has the same specifications they have different prices. 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 Flyswatter2, 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 2.1. On top of that, this JTAG debugger has a good clock frequency (up to 30MHz) for its price.

2.3.2.4 Software Debuggers

A software debugger is needed in order to communicate between the JTAG adapter and the host PC. The feature of the debugging system may vary because if could be either limited by the software, by the JTAG adapter or by the capability of the target CPU. 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, Source-Point for ARM and IAR embedded workbench and OpenOCD. Over the all possibilities, the only one which is open-source is OpenOCD. Although it lacks some features like multi-core debugging, it is quite complete.

2.4 Evasion techniques

The problem of using emulator is that it produces a lots of artifacts. These frameworks are good for analyzing the general behavior of an application, but these emulators exposes so much artifacts, that malware with even with the most basic checks are capable of

JTAG Debuggers	**Support OMAP4**	**Support Cortex-A**	**Communications**	**JTAG clock (MHz)**	**Download speed (KB/s)**	**RTCK**	**Support 1.8V**	**Price (€)**
IAR I-Jet	✗	✓	2.0 HS	32	1024	✓	✓	270.6
CrossConnect Pro	✗	✓	2.0 HS	✗	✗	✓	✗	339.92
Green Hills Probe	✓	✓	2.0 HS	✗	10240	✓	✓	✗
Segger J-link EDU	✗	✗	USB 2.0	15	1024	✓	✓	42
Segger J-link	✗	✓	USB 2.0	15	1024	✓	✓	298
Segger J-link Plus	✗	✓	USB 2.0	15	1024	✓	✓	498
Segger J-link Ultra	✗	✓	USB 2.0	15	3072	✓	✓	598
Segger J-link Pro	✗	✓	USB 2.0 /100T	50	3	✓	✓	798
Riff-box	✓	✓	USB 2.0	✗	250	✓	✓	92.58
Arium LC-500Se	✓	✓	USB 2.0 /100T	✗	✗	✓	✓	✗
H-JTAG Standard	✗	ARMv7	2.0HS	15	550	✗	✓	135.54
H-JTAG Professional	✗	ARMv7	2.0HS	15	550	✗	✓	356.75
ZY1000	✗	✓	100T	32	✗	✓	✗	1272.93
Lauterbach JTAG Debugger	✓	✓	✗	100	✗	✗	✓	≈1800
Abatron BDI 3000	✗	✗	100T	32	1500	✗	✓	✗
Flyswatter2	✓	✓	2.0 HS	30	✗	✓	✓	75.6
Bus Blaster	✓	✓	2.0 HS	1	✗	✗	✓	40

TABLE 2.2: JTAG Debuggers Comparison. This table shows a variety of boards and feature and some of them are mandatory for OpenST

	✓	The debugger has this feature
	✗	The debugger doesn't have this feature or it is not possible to know
Legend	2.0 HS	USB 2.0 High-speed (480Mbps)
	100T	Ethernet 100-Base T
	1000T	Ethernet 1000-Base T

Trace Hardware	**Support OMAP4**	**Support Cortex-A**	**Communications**	**JTAG clock (MHz)**	**RTCK**	**ETM clock (MHz)**	**Memory (MB)**	**Price (€)**
IAR I-jet Trace	✗	✗	2.0 HS /3.0	100	✓	150	✗	1274.9
Green Hills Super Probe	✓	✓	2.0 HS /1000T	✗	✓	1200	4096	✗
Segger J-Trace	✗	ARMv7	USB 2.0	12	✓	200	2	995
Arium LX-1000e	✓	✓	USB 2.0 /1000T	40	✗	680	2048	✗
Lauterbach Cortex Trace	✓	✓	✗	✗	✗	✗	4096	✗

TABLE 2.3: Trace Hardware Comparison

	✓	The debugger has this feature
	✗	The debugger doesn't have this feature or it is not possible to know
Legend	2.0 HS	USB 2.0 High-speed (480Mbps)
	3.0	USB 3.0 interface (5 Gbps)
	100T	Ethernet 100-Base T

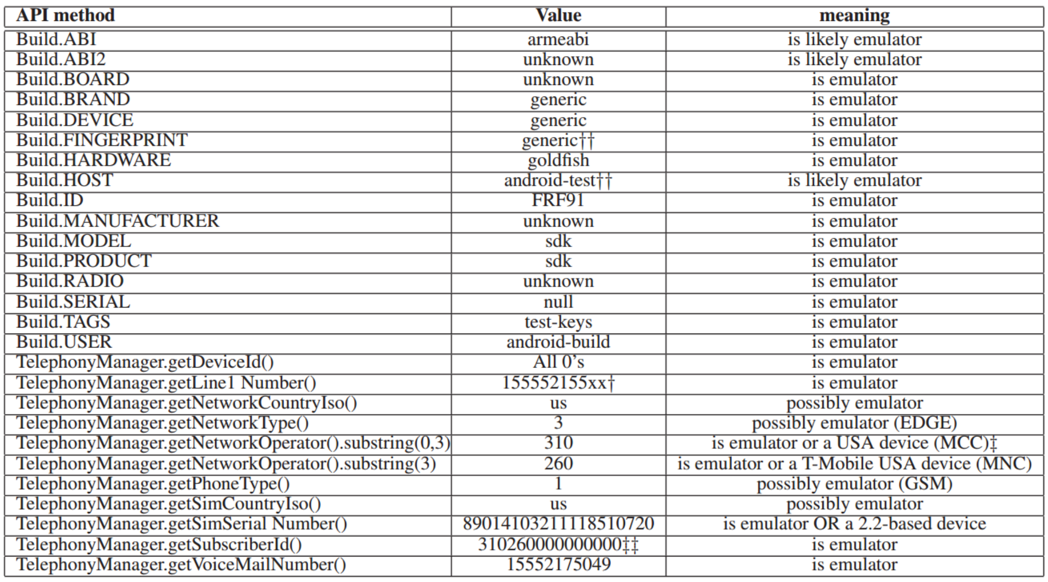
detecting the sandbox, thus hiding his true malicious behavior and thus pass the malware check.

Most Anti-VM techniques [20, 21] are based on simple heuristics which can be subdivided into three main categories: a) *static- heuristic* which checks on parameter that never changes its values, *dynamic heuristics* which checks the behavior of various sensor present on the device. and *hypervisor heuristics* which checks.

2.4.1 Static Heuristics

The static heuristics check for content which are usually present in real devices, such as serial number, and build version or the layout of the routing table.

Serial number Each smart-phone contains an International Mobile Station Equipment Identity (IMEI), which is an unique number which are able to identify a device over the



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 2.5: A list of known API call which can be used by malware to spot the presence of an emulator.

GSM network. Other identification number is International Mobile Subscriber Identity (IMSI), which is associated with the SIM card found in the phone.

Current build The value of the current build is stored into the system properties. For instance, the Android SDK provides the public class Build, which contain fields such as *PRODUCT*, *MODEL* and *HARDWARE* that can bee examined in order to determine if an application is running inside an emulated environment.

Routing table. An emulated Android device by default runs behind a virtual router within the 10.0.2.0/24 address space and its own IP address configured to 10.0.2.15. So its possible to exploit this information to detect the emulated environment.

Heuristics is not only limited to these three values. Figure 2.5) shows a list of API which can be used by an malware to detect the presence of an emulator.

2.4.2 Dynamic Heuristics

Even the most economic Android smart- phone incorporate sensors such an accelerometer, gyroscope, GPS, compass. Since these sensors collect information through the environment surrounding them, thus is possible to analyze the distribution of the output of these sensors to the existence of an emulator. In practice any of these sensors is not simulated in the Android emulator, or partially simulated by providing constant values. Thus in real world case a simple checks are enough.

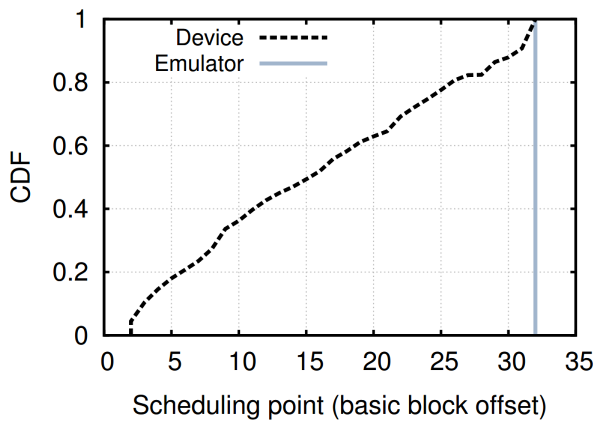


Figure 2.6: Due to optimizations, QEMU does not update the virtual PC on every instruction execution, and therefore many of the scheduling events that can take place are not exhibited on an emulated environment.

2.4.3 Hypervisor Heuristics

With errors in hardware design, such as CPU bugs, a complete emulation is a tough task. Not only the emulator don't emulate CPU bugs but it also does some optimization that are not present in real hardware.

QEMU scheduling QEMU is an efficient emulator for Android thus there are some artifact that it exposes. One of these is its scheduling, and the fact that QEMU does not update the virtual program counter (PC) at every instruction. Since QEMU translate the code in blocks it only increment the PC only when the normal execution flow encounter a branch, then the program counter is incremented and the context switch can occur. by checking the context switch occurred its possible to tell the difference between a real smart-phone and an emulator as shown in Figure 2.6).

2.5 Bare-metal Analysis

BareBox [22] is a tool that is able to perform efficient dynamic malware analysis on bare-metal system on commodity hardware. it's architecture is composed of Meta-OS which is responsible of saving and restoring the memory and the disk content in order to speed up the malware analysis without the need of rebooting a layout of the architecture is shown in Figure 2.7).

In the x86 architecture, restoring the physical memory becomes challenging because it involves overwriting both the GDT table and the page table, which, in turn, are used to translate virtual addresses into physical memory locations. This includes the current location (virtual address) of the code (EIP) that performs the memory restoring. That is, the memory restore code would change memory mappings that interfere with its own execution. Because of these circular dependencies, it is impossible to restore physical memory of a live operating system from within the same operating system (with arbitrary physical memory content). For this reason a small operating system, called Meta-OS has

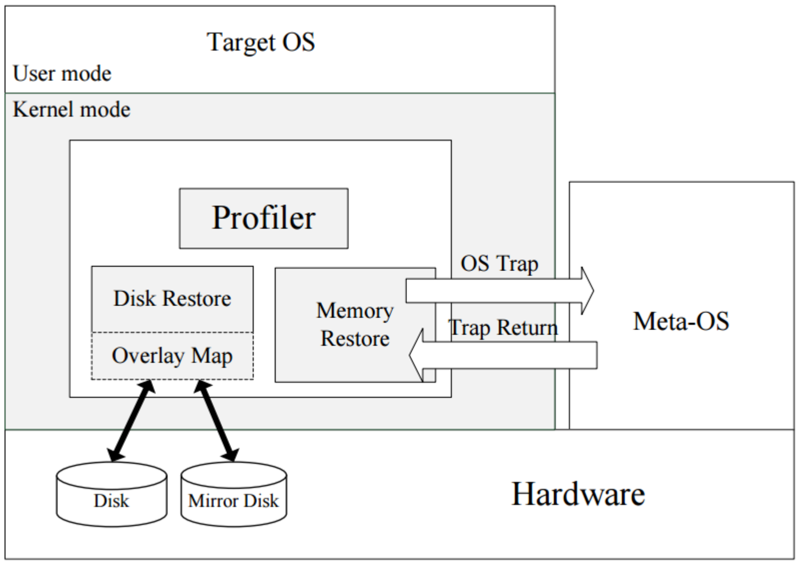


FIGURE 2.7: Architecture Overview of Barebox. (Left) target-OS which the analysis is focus on. (Right) Meta-OS which is the OS responsible for the introspection.

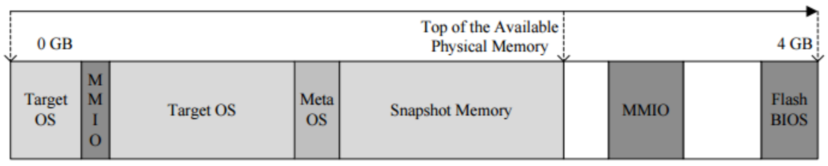


FIGURE 2.8: Physical memory allocation are divided in four sections: The target OS memory, the Meta-OS memory, the snapshot memory, and the hardware-reserved memory section

been implemented, as a memory restore component that resides outside of the physical memory of the target OS.

As the analysis begin, whenever a snapshot-save and snapshot- restore operations is triggered from the target-OS, a context switch is executed and the control is transferred to Meta-OS. 1) If a snapshot- save is triggered, the GDT table and the IDT table is save and the snapshot of both the memory and the disk is taken. 2) If a snapshot- restore is triggered, the GDT table and the IDT table of the target-OS is overwritten and their pointer stored in GDTR and IDTR CPU registers. The load operation is executed using IA-32 LGDT and LIDT instruction.

Physical Memory In order to take the available physical memory is partitioned into three parts. The operating system is loaded into the first part, which starts from the absolute hardware address zero. The second part of the physical memory is used to take a snapshot of the first part. Finally, the small operating system Meta- OS resides in the third part of the physical memory. This component is implemented as a kernel module that is loaded into the target OS (see Figure 2.8).

Disk Restore Whenever a snapshot-restore is issued the state of the disk is restored to that particular snapshot point. This is achieved by proper redirections of read and

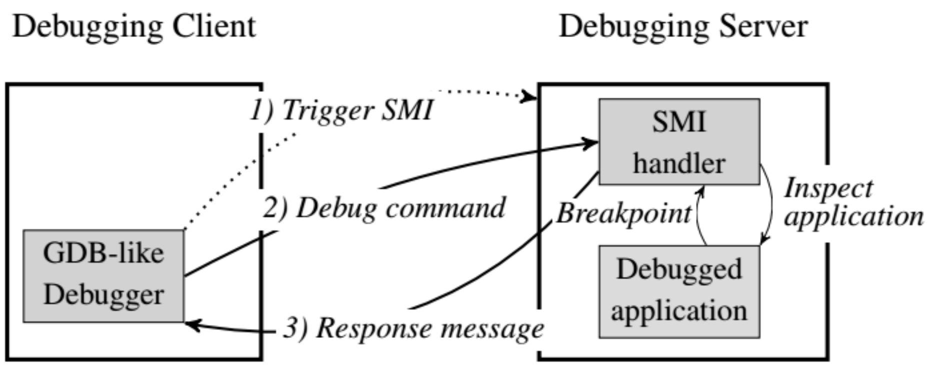


Figure 2.9: MalT architecture: (left) is shown an GDB debugger that communicate though serial interface with (right) MalT GDB server

write operations to the main and the mirror disks. A snapshot-save operation simply implies that all further write operations to the main disk are redirected to the mirror disk. With this redirection in place, Meta-OS effectively freeze the contents of the main disk. However, all read operations are still forwarded to the main disk, except those read operations to particular sectors that were previously redirected and written to the mirror disk. This is implemented two methods for the storage of the mirror disk; a RAM disk and a physical hard disk. This approach to physical memory and disk restoration makes the reboot-less restore of a bare-metal system possible.

MalT [23] is a tool that is able to perform efficient dynamic analysis at ring-2 level. It uses a novel approach that progresses toward stealthy debugging by leveraging System Management Mode (SMM) to transparently debug software on bare- metal. SMM is a mode of execution similar to Real and Protected modes available on x86 platforms. It provides a transparent mechanism for implementing platform-specific system control functions such as power management.

Figure 2.9 shows the architecture of the MalT system. The debugging client first sends a System Management Interrupt (SMI) triggering message to the debugging server; then reroute a serial interrupt to generate an SMI when the message is received. Secondly, once the debugging server enters SMM, the debugging client starts to send debugging commands to the SMI handler on the server. Thirdly, the SMI handler transparently executes the requested commands (e.g., list registers and set breakpoints) and sends a response message back to the client.

As with VMI system, MalT suffer from a semantic gap problem. In short the SMM cannot understand the semantic of raw memory. In order to resolve this issue MalT parses the EProcess structure which is a process descriptor containing crucial information about the current process (e.g., PID, process name).

In conclusion all this process since it is done in the SMM mode it is undetectable by any root-kit or hypervisor installed in the target machine. The only way the malware has to elude the introspection is by altering its process descriptor.

The goal of OpensST is to combine the advantages of CopperDroid, DroidScope and the idea of using bare-metal analysis such as MalT and BareBox. Thus OpenST is a cheap tool for Android malware analysis able to perform VMI and being based on unmodified physical hardware.

2.6 Limitations

The major drawback related to the VMI-based approach is that its all related to the accurate the emulator is, compared to the real hardware. In fact many emulators that perform malware analysis suffer from severe inconsistency which are able to reveal its presence to the malware therefore hiding its true malicious behavior and unvalidated the analysis. While a major limitation for bare-metal approach exposed here is that it uses special x86 mode to operate and its not portable to the ARM-based devices.

2.7 Goals and Challenges

The goal here is to create a open-source tool capable of performing dynamic analysis on hardware level. The tool should allow a security specialist to use techniques such as VMI to analyze the behavior of malicious application without exposing the drawback of using an VMI- based approach. Such as exposing artifacts such that a malware can be detect it. The tool developed here is called OpenST (Open SiliTracer), first effort into this direction. OpenST analyze malware automatically extract of its OS-level behavior. All without changing the original Android code nor tamping inserting introspection code into the target board. In practice takes the advantages of both the bare-metal and VMI-based approach.

The goal should be reached by respecting constrain such as budget. The trade-off between speed and price has been taken into account in the choice of the adapter. So for this work, the choice of the adapter was biased toward the cost-effective solutions. Given the information that we know we have chosen the Flyswatter2 adapter because it is one of the cheapest debuggers analyzed that has a decent frequency of 30MHz.

Given the use of low budget device, it is expected that the speed is limited. This can be partially dealt with more efficient that in the next release of this tool.

Chapter 3

Approach

In this chapter I present the overall approach of OpenST, focusing on the high-level workflow and functionality.

From an I/O point of view, as summarized in Figure 3.1. OpenST takes the target program to be traced and the operating system's kernel image as input. From this it produces a system call trace of the program. We divide OpenST's workflow into two, broad phases, **Phase 1 (Code Generation)**, which is run before the actual analysis, and **Phase 2 (Tracing)**, which is run during the program tracing.

I postpone the technical implementation details to Chapter 4.

3.1 Phase 1: Code Generation

This phase takes the operating system's kernel image as input and generates in output a C file containing the introspection procedure. The *introspection procedure* is a function that is called every time a system call invocation is intercepted at runtime: It is responsible for tracing it and its arguments. For example, if OpenST encounters a `open(‘‘foo.txt’’)` invocation, the introspection procedure will detect that an `open` system call has been invoked by user space process being monitored, with a string pointer to `‘‘foo.txt’’` passed as argument.

This phase is divided into **Phase 1.1 (System Call Prototypes and Data Reconstruction)** and **Phase 1.2 (Introspection Procedure Generation)** as shown in Figure 3.2. **Phase 1.1**, by parsing the kernel binary image (with debugging symbols), generates a file containing the information necessary to capture the invocation of system calls along with their arguments. This includes, for instance, data-types definitions and function prototypes. **Phase 1.2** then reads this information and generates the aforementioned introspection procedures.

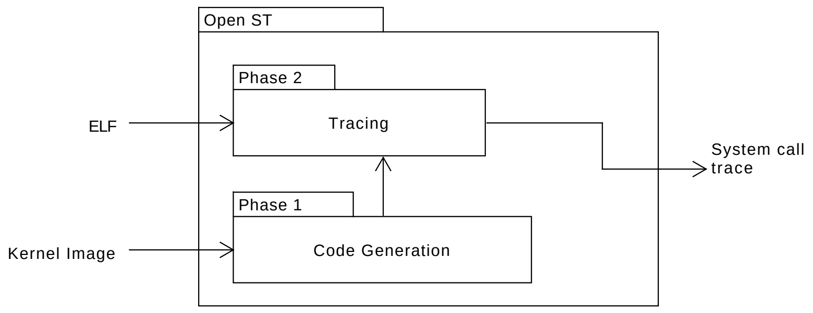


FIGURE 3.1: OpenST's logical overview. OpenST takes the target program to be traced and the operating system's kernel image as input and produce a system call trace in output. **Phase 1 (Code Generation)** is run before the actual analysis, and **Phase 2 (Tracing)** is responsible to perform the actual tracing.

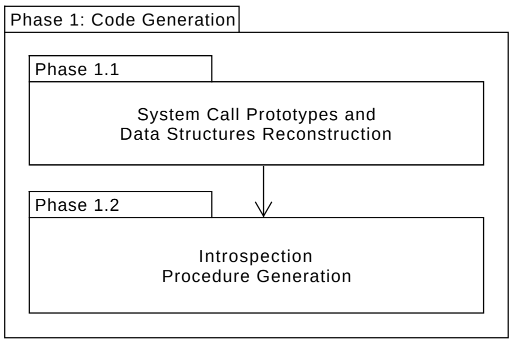

Phase 1: Code Generation
Phase 1.1
System Call Prototypes and Data Structures Reconstruction
Phase 1.2
Introspection Procedure Generation

FIGURE 3.2: Phase 1 takes the operating system's kernel image as input and generates in output a C file containing the introspection procedure.

3.1.1 Phase 1.1: System Call Prototypes and Data Structures Reconstruction

To reconstruct the system call and data structure definitions, this phase parses the meta-data left in the kernel image by the compiler. For example, in a 32-bits machine, given a data structure named `st1` with two integer fields and a character field

```c
struct st1 {
  int a;
  int b;
  char c;
};
```

the meta-data information would be that there is one structure with two consecutive 32-bits fields followed by an 8-bits field. The meta- data also includes information about the offset of the members of each structure. A similar example can be made for the definition of a system call. The output of this phase is a parsable version of the extracted meta-data.

In principle, the data structure information could be parsed from the kernel's source code. OpenST is agnostic in this regard, as long as the symbols and the data structures are reconstructed. However, in our implementation we prefer not to assume the availability

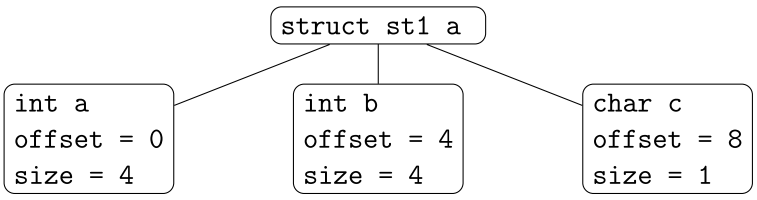


FIGURE 3.3: Example data structure internal representation extracted by **Phase 1.1** for a sample data structure, `st1`, with two integer fields and a character field.

of the kernel source code. This decision does not affect in any way the outcome of **Phase 1.1**.

3.1.2 Phase 1.2: Introspection Procedure Generation

To generate the introspection procedure, this phase first reconstructs the abstract syntax tree (AST) from the output of the previous phase. For instance, the AST of the `st1` data structure mentioned in the above example is depicted in Figure 3.3.

Secondly, for each structure and function definition present in the AST, it generates C code that dumps the content of the memory according to such definition, using the offsets and a base memory address.

Last, OpenST generates the introspection procedures in a form of library, one per system function. The generated library contains all the code necessary to dump a system call and the values of its arguments according to the respective data structures.

3.2 Phase 2: Tracing

This phase takes as input an executable binary and the output of **Phase 1** and generates as output a log file containing the list of system calls made by the program.

This phase is composed of 5 sub-phases as shown in Figure 3.2. **Phase 2.1** is responsible of managing the breakpoints that are needed to intercept the invocation of each system call. **Phase 2.2**, **Phase 2.3**, and **Phase 2.4** are responsible of reconstructing the system call, while Phase 2.5 takes the data generated by the previous phases and creates a human-readable and machine-readable log.

3.2.1 Phase 2.1: Hardware Breakpoint Management

This phase is responsible of setting and unsetting hardware breakpoints in the target's CPU. In fact it can set different type of breakpoint in order to trace every system call.

Two types of breakpoint are available. The first kind of breakpoint will halt the CPU at certain address. For example given the breakpoint at address `0x12345678`, the CPU will halt every-time the program counter match that value. The second kind of breakpoint will halt the CPU whenever a program execute an instruction and the context ID match the one specified in the of the current process. For example the program `runme` has context ID `0x12341234` if we set a context ID with breakpoint whenever it try to execute some functions the CPU will halt. The Context ID provides a mechanism to identify the currently executing process in multi-tasking environments. OpenST can also set a hybrid breakpoint which is just a combination of the first and second kind of breakpoint.

3.2.2 Phase 2.2: System Call Tapping

System Call Tapping is the handler which is called whenever the CPU halts.

For example given a program that has halted due to a breakpoint. This is handler will issue the Phase 2.3. If the reconstructed process during the phase 2.3 match the process we wanted to trace then the phase 2.4, 2.5 will be executed. Phase 2.4, 2.5 reconstruct the current system call and its arguments and output to a log.

3.2.3 Phase 2.3: Process Data Structure Reconstruction

This Phase is responsible of retrieving information about the current process. These information include process identifier (PID), the thread group identification (TGID) and the executable name (COMM) which could be used to filter out unwanted process.

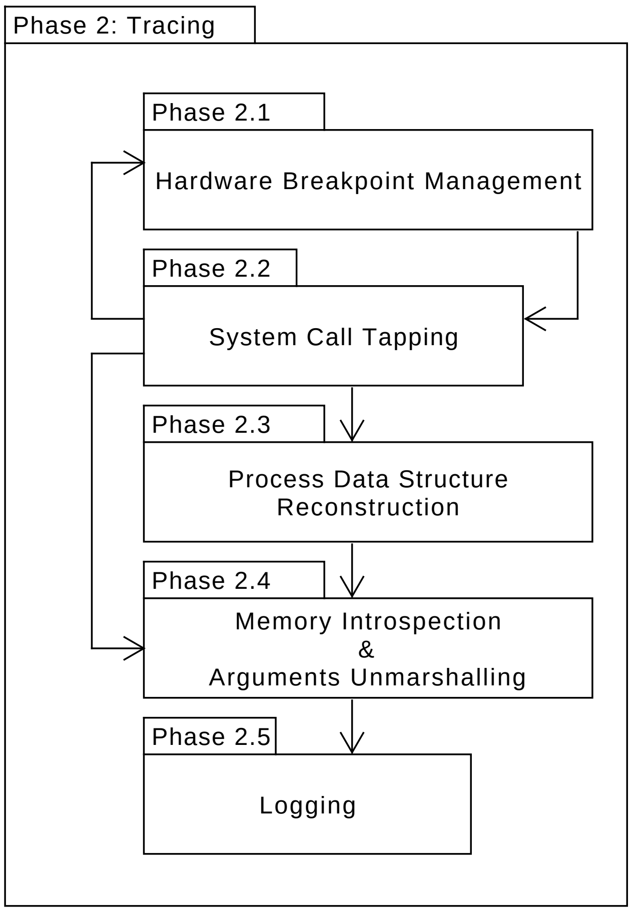

Phase 2: Tracing
Phase 2.1
Hardware Breakpoint Management
Phase 2.2
System Call Tapping
Phase 2.3
Process Data Structure Reconstruction
Phase 2.4
Memory Introspection & Arguments Unmarshalling
Phase 2.5
Logging

Figure 3.4: Phase 2

For example, we are tracing a program named `runme`. During the reconstruction of information about the current process If we found the `COMM` is different than `runme` then the Phase 2.4 and 2.5 wont be executed otherwise it will.

3.2.4 Phase 2.4: Memory Introspection

This Phase use the library generated in Phase 1.2 to introspect the memory in order to reconstruct the system call and its arguments.

For example given the previous program. OpenST call `dump_sys_foo(depth, handler)`. If depth equals to 0 then `sys_foo(0x12345678)` is produced. If depth greater than 0 `sys_foo(12, 34, 'a')` is produced.

3.2.5 Phase 2.5: Logging

Logging in OpenST is a series of `#define` in order to improve the legibility. Reconstructed system call are shown in terminal. But a log file is also made available for later examination.

Chapter 4

Implementation details

OpenST's implementation details is explained in this chapter. This chapter is divided in two sections again, but this time I will tank about its technical details, Phase 1: Code Generation which will talk the programs we used to generate the Introspection code. Phase 2: Tracing which will talk about the program that does the tracing.

4.1 Phase 1: Code Generation

This phase is responsible of generating the necessary code for the introspection. This can be obtained by parsing the kernel source code or the the debugging information associated with the compiled kernel. This information includes system call procedure definition and its argument structures. By using this information and knowing the API of OpenOCD. OpenST is able to produce a library capable of introspecting the target machine. This Phase is described more in detail in Section 4.1.1 Phase 1.1 and Section 4.1.2 Phase 1.2.

4.1.1 Phase 1.1: System Call Prototype and Structures Reconstruction

This phase the kernel binary and parses the DWARF format in order to reconstruct the system call prototype and its arguments. Debugging with attributed record formats (DWARF) is a debugging file format used by many compilers and debuggers to support source-level debugging. It is the format of debugging information within an object file. The DWARF description of a program is a tree structure where each node can have children or siblings. The nodes might represent types, variables, or functions.

For example given this program as input after being compiled with gcc.

```c
struct st1 {
   int a;
   int b;
   char c;
};

void sys_foo(struct st1 *bar){}
```

Listing 4.1: hello.c

DWARF are represented in the following tree form after being parsed.

```
Number TAG (0x0)
 1      DW_TAG_compile_unit    [has children]
  DW_AT_producer     DW_FORM_strp
  DW_AT_language     DW_FORM_data1
  DW_AT_name         DW_FORM_strp
  DW_AT_comp_dir     DW_FORM_strp
  DW_AT_low_pc       DW_FORM_addr
  DW_AT_high_pc      DW_FORM_data8
  DW_AT_stmt_list    DW_FORM_sec_offset
  DW_AT value: 0     DW_FORM value: 0
 2      DW_TAG_structure_type    [has children]
  DW_AT_name         DW_FORM_string
  DW_AT_byte_size    DW_FORM_data1
  DW_AT_decl_file    DW_FORM_data1
  DW_AT_decl_line    DW_FORM_data1
  DW_AT_sibling      DW_FORM_ref4
  DW_AT value: 0     DW_FORM value: 0
 3      DW_TAG_member    [no children]
  DW_AT_name         DW_FORM_string
  DW_AT_decl_file    DW_FORM_data1
  DW_AT_decl_line    DW_FORM_data1
  DW_AT_type         DW_FORM_ref4
  DW_AT_data_member_location DW_FORM_data1
  DW_AT value: 0     DW_FORM value: 0
 4      DW_TAG_base_type    [no children]
  DW_AT_byte_size    DW_FORM_data1
  DW_AT_encoding     DW_FORM_data1
  DW_AT_name         DW_FORM_string
  DW_AT value: 0     DW_FORM value: 0
 5      DW_TAG_base_type    [no children]
  DW_AT_byte_size    DW_FORM_data1
  DW_AT_encoding     DW_FORM_data1
  DW_AT_name         DW_FORM_strp
  DW_AT value: 0     DW_FORM value: 0
 6      DW_TAG_subprogram    [has children]
  DW_AT_external     DW_FORM_flag_present
  DW_AT_name         DW_FORM_strp
  DW_AT_decl_file    DW_FORM_data1
  DW_AT_decl_line    DW_FORM_data1
  DW_AT_prototyped   DW_FORM_flag_present
  DW_AT_low_pc       DW_FORM_addr
  DW_AT_high_pc      DW_FORM_data8
  DW_AT_frame_base   DW_FORM_exprloc
  DW_AT_GNU_all_call_sites DW_FORM_flag_present
  DW_AT_sibling      DW_FORM_ref4
  DW_AT value: 0     DW_FORM value: 0
 7      DW_TAG_formal_parameter    [no children]
  DW_AT_name         DW_FORM_string
  DW_AT_decl_file    DW_FORM_data1
  DW_AT_decl_line    DW_FORM_data1
  DW_AT_type         DW_FORM_ref4
  DW_AT_location     DW_FORM_exprloc
  DW_AT value: 0     DW_FORM value: 0
 8      DW_TAG_pointer_type    [no children]
  DW_AT_byte_size    DW_FORM_data1
  DW_AT_type         DW_FORM_ref4
  DW_AT value: 0     DW_FORM value: 0
```

Listing 4.2: DWARF format

Using that information we are able to reconstruct the source code of the program.

4.1.2 Phase 1.2: Introspection Procedure Generation

OpenST uses pycparser to generate its introspection code. Pycparser is a parser for the C language, written in Python and aim to support full C99 languages, some feature from C11 might be supported. Essentially we use pycparser to parse the information gained in the Phase 1.1. All these information will be stored into an AST of C99 grammar. In addition information non present in the grammar is stored as attribute into the node(Eg. size and offset).

The generation of the introspection code start from inspecting the syscalls. Suppose we want to dump sys_clock_gettime.

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

Listing 4.3: clock_gettime system call prototype

```
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 4.4: Structures and typedefs used on clock_gettime

The output procedure will be called dump_sys_clock_gettime.

```
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]);
```

```c
    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 4.5: Memory introspection code of clock_gettime

The procedure dump_sys_clock_gettime will return a string containing all the information related to the dump. The dumped_params will be an array of strings which hold all the output of procedures called by dump_sys_clock_gettime then all these output are merged into param_str. In that case when we find a basic type in the arguments in the system call we use get_uint32_t_register_by_name procedure to get its value, this function the register through JTAG. Otherwise in the case we encounter a non basic type so i call the function dump_timespec.

```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 4.6: Dump structure timespec

This function is only executed if depth is greater than 0. The function dump_timespec works the same way as dump_sys_clock_gettime, so it will dump tv_sec as an integer because pycparser is able to de-reference the typedef, but since sec and nsec its values are stored in memory, I call dump_long_int_from_mem.

```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 4.7: Dump long integer from memory

Then dump_long_int_from_mem will read from memory using a JTAG function and then using dump_generic

```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 4.8: Dump generic type's value

to write into the param_str and then return. The dump_int:

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

Listing 4.9: Dump integer from register

is only responsible to print to a string, while the dump_prt presents and prints in hexadecimal format 0x;

4.1.2.1 Challenges

During the writing of the pycparser we encounter these challenges:

- Recursive dumping of each structure
- Translation of typedef data types
- Meta-data of anonymous structure

Structure might have itself as member(this might happen in a list) so the dumping program might not end or crash, that's why we exit the dump as soon as we encounter a not valid pointer. In the typedef case we needed to preprocessor it. In the last case we might have anonymous structure inside a structure. That's why we use hash to check if a structure is already defined or not, in order to avoid duplication of code.

4.1.2.2 Conclusion

To conclude the `dump_sys...` main procedure which is being called and which return a string contain all the information introspected and unmarshalled. This is done modularly because `dump_sys_clock_gettime` will also call `dump_timespec` which is a structure created to dump a specific structure.

4.2 Phase 2: Tracing

This section talks about how we analyze an ELF file with OpenST. OpenST uses OpenOCD as the main tool for handling the JTAG communication between the host and the target machine, through a JTAG probe. In the subsections I will explain how I changed the OpenOCD to suit my need.

4.2.1 Phase 2.1: Hardware Breakpoint Management

In ARM based processor its possible to issue system call like "int 80" in x86 by using the SWI assembly instruction. This instruction will generate a software interrupt exception and therefore the PC of the CPU will jump to the exception vector table. OpenST will then issue the breakpoint into the exception vector table in order to halt the execution when the SWI is executed. Then the control flow of the program is passed to the Phase 2.2. After the Phase 2.2 has finished, operations needed to resume the normal flow of the program will be done. To do so we need to remove the breakpoint at the current PC and make a new breakpoint at PC + 4 and then resume. Thus the stepping is simulate in that way. Here is the implementation:

```
/* 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;

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)
     ;
} 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)
     ;
}
```

Listing 4.10: Implementation of Hardware Breakpoint Management

As we can note we can set a context ID in order to track a specific process. The cortex ID can be obtained by using the MRC instruction.

4.2.2 Phase 2.2: System Call Tapping

System Call Tapping is the handler which waits for the CPU to reach interrupt exception vector during that time the CPU is halted and phase 2.3 and 2.4 is called. In OpenOCD the callback function is implemented in that way:

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

Listing 4.11: Implementation of Hardware Breakpoint Management

4.2.3 Phase 2.3: Process Data Structure Reconstruction

In this Phase we want to introspect the kernel memory in order to retrieve information about the calling process. To do so we need to analyze the process descriptor. The process descriptor is stored in a struct named task_struct. This struct contain information about the PID, TGID and COMM. The following code will show how we retrieve the this information.

```
    /* 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 4.12: Process Data Reconstruction

In order to get the task_struct we need to get the thread_info which is stored in the end of the kernel. Thread_info is stored in the end of the stack pointer. The source code of thread_info:

```
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;
    __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 4.13: Pahole output after instrumentation

We used pahole to get its size, which is 8,192byte. Thus we put a mask to the SP in order to get the thread info address.

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

Listing 4.14: Pahole output after instrumentation

In conclusion this is needed in order to inform the Phase 2.1 about the context ID of the current process.

4.2.4 Phase 2.4: Memory Introspection

The concept here is that we an array of functions. These function will dump the system call associated with it and return a string with all its information.

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

Listing 4.15: Declaration of the array of function pointers

The following function map the array of function to the right system call generated in the Phase 1.

```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 4.16: Populate the array of function pointers

In the end, we just need to call sys_ptr[“syscall number”] to get the dump of the syscall and have a string containing its dump.

```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 4.17: System calls logging

4.2.5 Phase 2.5: Logging

OpenST log the result with fprintf as shown in Listing 4.17. In addition a real-time log is possible with LOG_SYSCALL macro.

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

Listing 4.18: LOG_SYSCALL macro definition

This macro facilitate the logging with the right color for console the output.

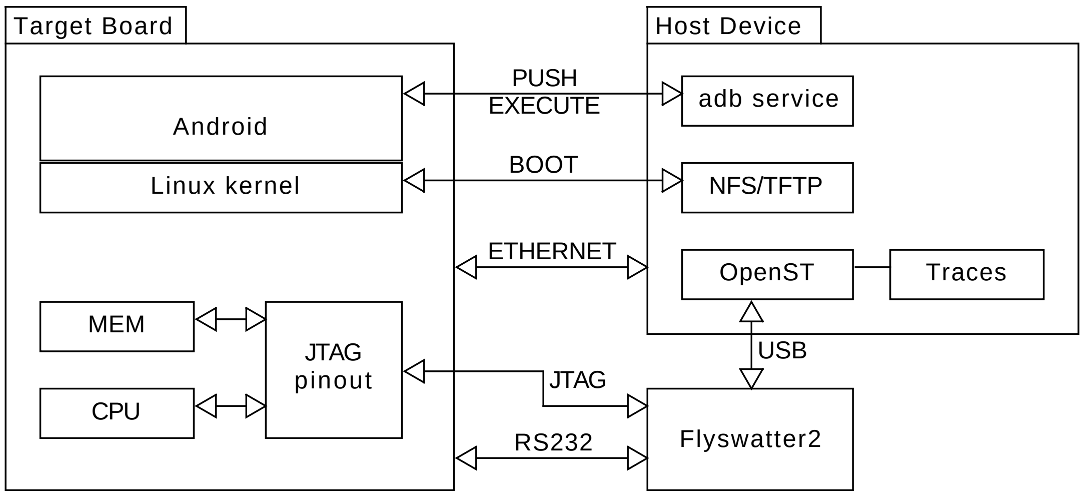


Figure 4.1: OpenST architecture and connection are shown: (left) the target board and the component involved is shown. (Right) both the JTAG adapter and the host devices are shown.

4.3 Technical Details and Prototype Architecture

In this chapter we'll discuss about the system details and architecture of OpenST and the hardware devices involved.

4.3.1 System Architecture

In this section I will talk about the software stack we use in order to boot. And how devices are interconnected. Figure 4.1 show the architecture we use for our experiments:

4.3.1.1 Debugging Architecture

To automate the some of the process we have employed OpenOCD, NFS server, TFTP server and adb server. OpenOCD is the tool responsible of the communication with the JTAG adapter. While we use TFTP to upload the kernel image into the pandaboard, through BOOTP bootloader, This is possible because we have compiled U-Boot to do so. Then U-Boot will boot the kernel with the NFS to out host device. The NFS server hold the filesystem used by the Pandaboard. While adb server is needed in order to install an APK and to execute it. All these automation is done through the tcl language which OpenOCD rely on.

4.3.1.2 Booting Schema

One of the mandatory requirements is to reboot the board and restore the filesystem every time the a trace has been executed. As soon as the Pandaboard is turned on it will execute the internal Boot ROM. This code is flashed during the manufacturing

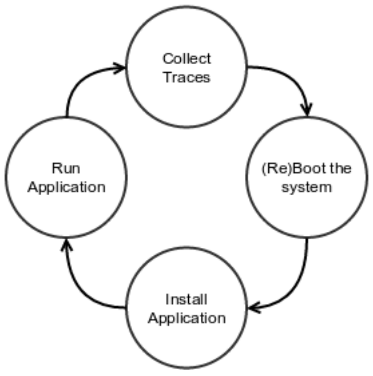


FIGURE 4.2: Approach used to automate OpenST. Various step of the restoration of the original system has been showed.

process and cannot be altered. The Boot ROM passes the Control to MLO. The fist level bootloader (MLO) will start UART and pass the control to U-Boot. U-Boot is the application which passes control to the Linux system. The main goal of U-Boot is to retrieve the Linux kernel and provide the location of the Linux filesystem to it. In our case the filesystem is in the NFS. U-Boot also load the vmlinux and tree.dtb into the memory tough TFTP. The tree.dtb is the file containing the device tree of our Pandaboard. The automation is shown in the Figure 4.2.

4.3.2 System Requirements

The overall system requirements is to build a system were every highly available and cheap. These component include a host device, a target board and a JTAG debugger.

4.3.2.1 Host Device

As host device anything that support NFS could be fine. But I recommend a server with x86 processor. Even though we used an ARM- based device(Raspberry Pi 2) to do the final testing.

4.3.2.2 Target Board

I use a pandaboard as developing board, as it supports Android, has an ARM-based CPU and RS232 and ethernet and JTAG pin-out. But any board with these I/O devices and that supports android would be fine. In particular the pandaboard has a OMAP4460 as its SoC which has two ARM Cortex-A9 as MPU. I disabled one of its kernel in the kernel because OpenOCD don't support SMP right now.

4.3.2.3 JTAG Debugger

There are a lot of devices capable of debugging and tracing for JTAG. I have chosen one of the cheapest device in the market. Cheap device usually comes with no custom software and it rely on open source alternative such as OpenOCD. And also because the Flyswatter2 has a decent frequency of 30MHz.

Chapter 5

Experimental Validation

In this chapter we present the results of our experimental evaluation. In summary, we performed macro- and micro-benchmarking and found out that OpenST imposes substantial overhead that prevents its usage in real-world settings. Interestingly, the state-of-the-art system, MALT [23], imposes a slowdown between 2 to 973 times, whereas OpenST imposes a slowdown of 70 times on average.

5.1 System setup

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. The JTAG debugger is Flyswatter2 which has clock frequency up to 30MHz. The target is a Pandaboard ES rev B3 with a dual-core ARM® Cortex™-A9 MPCore™with Symmetric Multiprocessing (SMP) at up to 1.2 GHz each.

5.2 Case Study

In this experiment OpenST is used to trace a series of system call issued by a running process. The running process is uploaded to the target board by adb (an utility of the android SDK). Adb has been integrated in OpenST in order to automate it. In order to upload the executable to the target machine the malware analysis have to connect to the listening socket on port 4444 by using telnet. After the connection has been established he can upload and trade the executable by performing "systrace bench *exec_name*" command. Any OS-specific behavior will be shown in real-time and dumped into a file for later analysis. The Figure 5.1 show the hardware component involved in this setup.

5.3 Correctness

We checked the correctness by creating a testing program which issues a set of known system calls and cross-checking the output of OpenST with the one that we expected. The test program is created in part in assembly and in part in C. The assembly code has

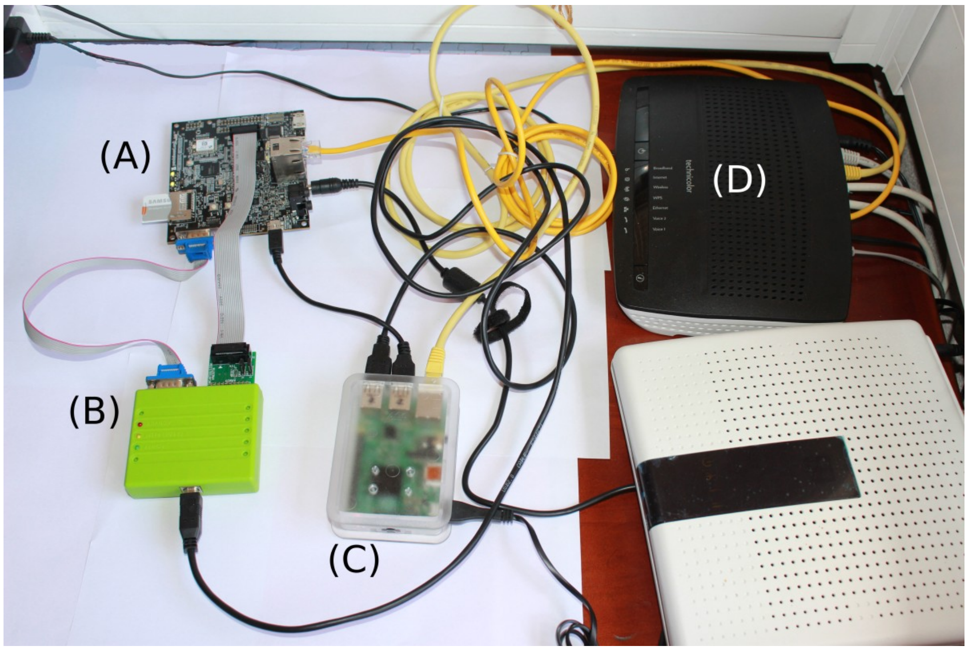


FIGURE 5.1: Different component involved in the system setup: (A) The target board. (B) JTAG adapter. (C) Host PC. (D) Used as switch.

implemented a procedure which call a system call which number is the first argument of the procedure. That system call is then called many times and arguments set to null. The C program is responsible of calling the assembly procedure with the right argument in order to issue known system calls. A log is then created after the program termination. A part of the trace output is shown here:

```
[pid:7618 tgid:7618 comm:micro_bench1] sys_open({0x0,0,0})
[pid:7618 tgid:7618 comm:micro_bench1] sys_write({1,0x4006e000,7})
[pid:7618 tgid:7618 comm:micro_bench1] sys_close(0)
[pid:7618 tgid:7618 comm:micro_bench1] sys_write({1,0x4006e000,5})
[pid:7618 tgid:7618 comm:micro_bench1] sys_unlink(0x0)
[pid:7618 tgid:7618 comm:micro_bench1] sys_write({1,0x4006e000,5})
[pid:7618 tgid:7618 comm:micro_bench1] sys_execve_wrapper({0x0,0x0,0x0,0x0})
[pid:7618 tgid:7618 comm:micro_bench1] sys_write({1,0x4006e000,5})
```

LISTING 5.1: Tracing output

In fact system calls: sys_write, sys_open, sys_close, sys_unlink and sys_execve_wrapper has been issue by our C program. The system call sys_write is used in our program to produce the log file containing timing measurement of the system calls.

5.4 Micro Benchmarks

The program used here is the same program we use for checking the correctness. The program is executed with and without instrumentation in order to obtain some statistics.

OpenST is run with 3 frequency levels (290kHz, 2.9MHz and 29MHz) in order to check the presence of bottleneck in the JTAG adapter. This can be done by deactivating

	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

Table 5.1: Micro benchmarking results: comparison between the instrumented and not instrumented system by varying the speed of the JTAG adapter.

		Execution time(s)		Slowdown
		Average Time	**Standard Deviation**	
7za	native	3.08	0.88	70.3 ± 16.5
	OpenST	204	19	
ps	native	1.26	0.35	119 ± 33
	OpenST	141	14	
netstat	native	0.023	0.0026	5667 ± 813
	OpenST	134	21.8	

Table 5.2: Macro benchmarking results: a comparison between the instrumented and not instrumented system on the execution time of three programs.

the adaptive clock and manually setting the clock of the adapter. This result of the experiment is shown in Table 5.1.

We came to the conclusion that the overhead of this approach, with this setup, is high. It seems, that even if we had a better JTAG adapter, the performance would not be much better. So, we presume that the bottleneck of this approach is the latency of the communication channel. In order to accurately measure time of each system call, we accessed the CPU ticks register(insert the register here) and multiplied that value by the max frequency of the target's CPU, so we can have the time. If we would use clock_gettime, which is a system call, we would lose precision on the benchmarks.

The rest results its shown in Figure 5.2. As we can see, we have almost a constant timing when the target is instrumented. This is due to the fact that we have almost the same operations, while the real system call execution have a negligible time. The slowdown is really high for simple system call, because we go from the nanoseconds order (native system call) to the microseconds order (instrumentation). In conclusion, we can see that the performance hit with a program that issues many system calls is really high. So, OpenST works best with programs that issues fewer system calls.

5.5 Macro Benchmarks

The macro benchmark will test several real-world programs (7zip, ps, netstat) in order to see its performance and compare its execution time with OpenST, without and the slowdown. This result in shown in Table 5.2

The overhead on real programs is also high, but it is way lower than the overhead at the micro level. The reason behind this results is that the programs have instructions that are not system calls, as opposed to the ones used at the micro level. Another thing that should be noted, is that we have a big performance hit in the begging of the analysis. The reason is that at that point, we still do not know the PID, and thus, the context ID of the process to be traced. So, during that time, we perform system-wide analysis until we encounter the out executable.

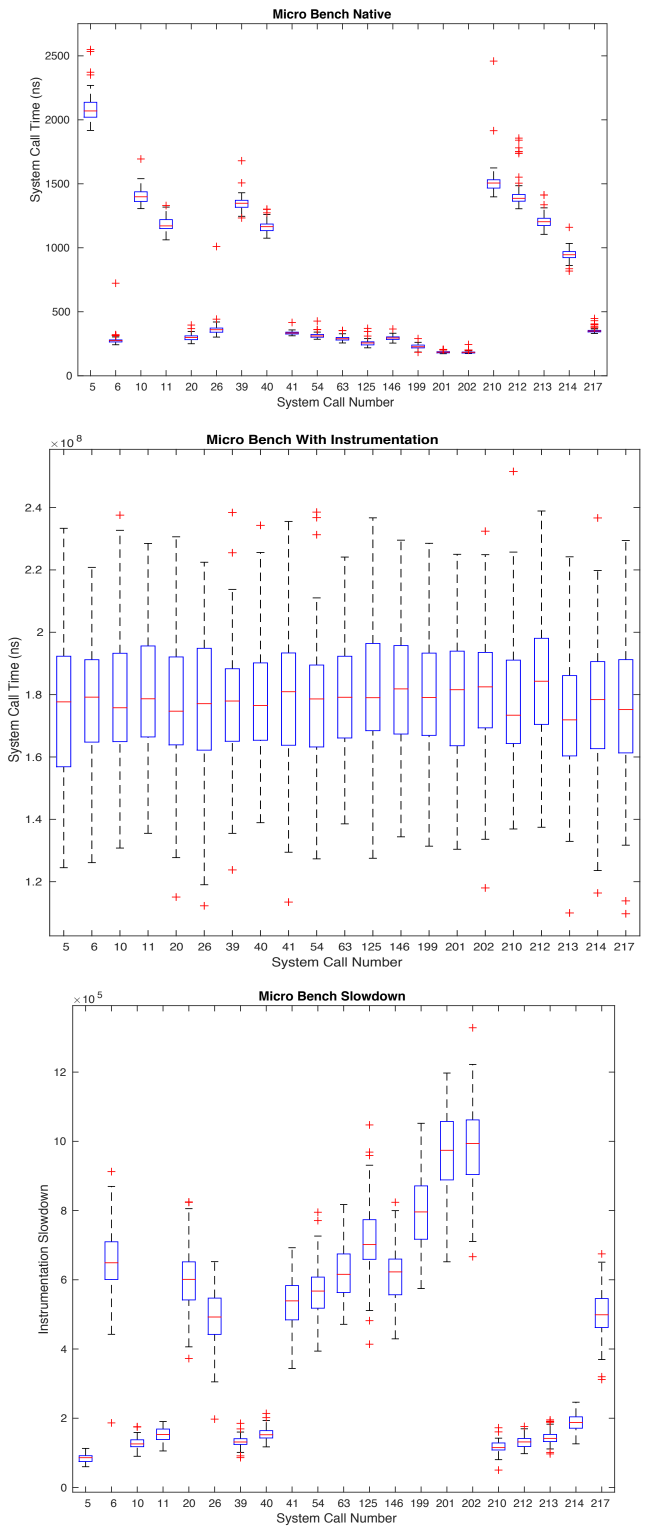


FIGURE 5.2: Micro benchmarking on the target: top figure shows the overhead of OpenST, without instrumentation. Middle figure shows the overhead of OpenST, with instrumentation. Bottom figure shows the slowdown of OpenST due to the instrumentation.

Chapter 6

Conclusions and Future Work

In this work we analyzed JTAG adapter available in the market and its cost effectiveness. JTAG adapter differs in cost and feature. Some of vendor provide closed-source software requiring extra money for license which we don't need. After scrolling the list of JTAG adapter and filtering the expensive ones, the flyswatter2 is the best choice since its not expensive and has all the basic feature we needed. While the pandaboard was chosen simply because was available in the lab.

Design and implementation of a system architecture for malicious application analysis required the study of different solutions. Given the novelty of this approach we adopted a bottom-up developing method in order to suit our needs, because we had to explore or test new software or approach and see if they integrate with the rest of the architecture. In fact not only we test new approach but we also try different variant in order to take the best. Also the integration of the sub-solution let us refine even more the software in term of correctness and robustness. In Section 3.1 Phase 1: Code Generation is the most complex part in the of testing due to the huge amount of code being generated. In 3.1.1 Phase 1.1 that reconstruct the system calls and argument structure are usually validated when we run the parser code of Phase 1.2. While the code generated by in Section 3.1.2 Phase 1.2 is validated by tracing a real world program. In the end we integrated adb in order to automate the analysis process. So sample are tested and result saved into a file, and the board rebooted for the next test.

Then we ran two types of benchmarks to highlight the bottlenecks in order to see where it can be improved and its limits. In fact in the micro-bench experiment we show how high is the overhead per system call, but this is due in part to the JTAG adapter an the off-chip communication latency. But the impact where smaller in the macro-bench because of the density of system calls not as high as in the micro- bench but still the overhear impact is still not negligible and improvement can be made possible by an extensive testing with other combination of JTAG adapter and board, or testing with an alternative high-performance interface (e.g., SWD). Lastly the tracing logic could be implemented into a FPAG using tools such as *PandA Project* [24] which can leverage the AMBA AHB a high-performance bus for communicating with the CPU and the memory.

OpenST has some limitations most of which can be surpassed by changing the hardware. The main limitation of OpenST is the slow performance. Slow performance derive from the slow link between the debug adapter and the target board as shown in Table 5.1 the maximum frequency of the Flyswatter2 is 30MHz. Even though we see an increment on the speed by incrementing the JTAG adapter. I presume the bottleneck could be in the latency of the JTAG. because it has to go out of the SoC and the overhead of the JTAG protocol, so its not negligible. Another limitation is the operation system scheduler, because in order to trace a program we need to find its context ID. And we can only find it if we set a system-wide breakpoint in order to catch the analyzed program and then we are able to trace it with hybrid-breakpoint. Lastly OpenOCD does not support SMP right not, so we have disabled the second core of the pandaboard.

To conclude OpenST show the concrete possibility of a new type of dynamic analysis though hardware. Even though is not as fast as other emulation-based solution, it has a lots of room for improvements. Still, OpenST can be used when other method fail to analyze.

Bibliography

[1] IDC. Smartphone os market share, 2014. URL `http://www.idc.com/prodserv/smartphone-os-market-share.jsp`.

[2] Fortinet. Fortiguard midyear threat report, 2013.

[3] Juniper Networks. Third annual mobile threats report, 2013.

[4] Eran Kalige and Darrel Burkey. A case study of eurograbber: How 36 million euros was stolen via malware. Technical report, 2013. URL `http://www.mtechpro.com/2013/mconnect/february/dyncontent/Eurograbber_White_Paper.pdf`.

[5] Google. Google bouncer, 2012. URL `http://googlemobile.blogspot.it/2012/02/android-and-security.html`.

[6] Xuxian Jiang. An evaluation of the application (”app”) verification service in android 4.2, 2014.

[7] Aubrey-Derrick Schmidt Seyit Ahmet Camtepe Thomas Blasing, Leonid Batyuk and Sahin Albayrak. An android application sandbox system for suspicious software detection. In *In Proceedings of the 5th International Conference on Malicious and Unwanted Software (MALWARE)*, 2010.

[8] Aristide Fattori Alessandro Reina and Lorenzo Cavallaro. A system callcentric analysis and stimulation technique to automatically reconstruct android malware behaviors. In *In Proceedings of the 6th European Workshop on System Security (EUROSEC)*, 2010.

[9] Byung-Gon Chunn Landon P. Cox Jaeyeon Jung Patrick McDaniel William Enck, Peter Gilbert and Anmol N. Sheth. Appsplayground: Automatic security analysis of smartphone applications. In *In Proceedings of the 3rd ACM conference on Data and Application Security and Privacy (CODASPY)*, 2013.

[10] Yan Chen Vaibhav Rastogi and William Enck. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *In Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[11] Florian Echtler Thomas Schreck Michael Spreitzenbarth, Felix Freiling and Johannes Hoffmann. Mobile-sandbox: Having a deeper look into android applications. In *In Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC)*, 2013.

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

[13] Fatemeh Azmandian, Micha Moffie, Malak Alshawabkeh, Jennifer Dy, Javed Aslam, and David Kaeli. Virtual machine monitor-based lightweight intrusion detection. *SIGOPS Oper. Syst. Rev.*, 45(2):38–53, July 2011. ISSN 0163-5980. doi: 10.1145/2007183.2007189. URL `http://doi.acm.org/10.1145/2007183.2007189`.

[14] Abhinav Srivastava and Jonathon Giffin. Tamper-resistant, application-aware blocking of malicious network connections. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, RAID '08, pages 39–58, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-87402-7. doi: 10.1007/978-3-540-87403-4_3. URL `http://dx.doi.org/10.1007/978-3-540-87403-4_3`.

[15] B.D. Payne, M.D.P. de Carbone, and Wenke Lee. Secure and flexible monitoring of virtual machines. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 385–397, Dec 2007. doi: 10.1109/ACSAC.2007.10.

[16] Hyun wook Baek, Abhinav Srivastava, and Jacobus Van der Merwe. Cloudvmi: Virtual machine introspection as a cloud service. In *Proceedings of the 2014 IEEE International Conference on Cloud Engineering*, IC2E '14, pages 153–158, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-3766-0. doi: 10.1109/IC2E.2014.82. URL `http://dx.doi.org/10.1109/IC2E.2014.82`.

[17] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *In Proc. Network and Distributed Systems Security Symposium*, pages 191–206, 2003.

[18] A. Fattori K. Tam, S. J. Khan and L. Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *Network and Distributed System Security (NDSS) Symposium*, San Diego, CA, USA, Feburary 2015.

[19] Dominic Rath. Open on-chip-debugger, 2005. URL `http://openocd.org/files/thesis.pdf`.

[20] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Rage against the virtual machine: Hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security*, EuroSec '14, pages 5:1–5:6, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2715-2. doi: 10.1145/2592791.2592796. URL `http://doi.acm.org/10.1145/2592791.2592796`.

[21] Timothy Vidas and Nicolas Christin. Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, pages 447–458, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2800-5. doi: 10.1145/2590296.2590325. URL `http://doi.acm.org/10.1145/2590296.2590325`.

[22] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. Barebox: Efficient malware analysis on bare-metal. In *Proceedings of the 27th Annual Computer*

Security Applications Conference, ACSAC '11, pages 403–412, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0672-0. doi: 10.1145/2076732.2076790. URL `http://doi.acm.org/10.1145/2076732.2076790`.

[23] Fengwei Zhang, Kevin Leach, Angelos Stavrou, Haining Wang, and Kun Sun. Using hardware features for increased debugging transparency. In *Proceedings of The 36th IEEE Symposium on Security and Privacy (S&P'15)*, May 2015.

[24] G. Kuzmanov, V.M. Sima, K. Bertels, J.G.F. de Coutinho, W. Luk, G. Marchiori, R. Tripiccione, and F. Ferrandi. hartes: Holistic approach to reconfigurable real-time embedded systems. *Reconfigurable Computing: From FPGAs to Hardware/Software Codesign*, page 91, 2011.