UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI

DOTTORATO DI RICERCA IN INFORMATICA
XXVI Ciclo

Discovering anomalous behaviors by advanced program analysis techniques

Relatore:	Prof. Danilo Mauro Bruschi
Correlatore:	Dr. Lorenzo Cavallaro
Coordinatore del Dottorato:	Prof. Ernesto Damiani

Tesi di:	Alessandro Reina
Matricola:	R09030

Anno Accademico 2012/2013

UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE
Cycle XXVI

Discovering anomalous behaviors by advanced program analysis techniques

Advisor:	Prof. Danilo Mauro Bruschi
Co-Advisor:	Dr. Lorenzo Cavallaro
PhD Coordinator:	Prof. Ernesto Damiani

PhD Candidate:	Alessandro Reina
ID:	R09030

Academic Year 2012/2013

Abstract of the dissertation

Discovering anomalous behaviors by advanced program analysis techniques

by
Alessandro Reina

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

Università degli Studi di Milano
2012/2013

As soon as a technology started to be used by the masses, ended up as a target of the investigation of *bad guys* that write malicious software with the only and explicit intent to damage users and take control of their systems to perform different types of fraud. Malicious programs, in fact, are a serious threat for the security and privacy of billions of users. The *bad guys* are the main characters of this unstoppable threat which improves as the time goes by. At the beginning it was pure computer vandalism, then turned into petty theft followed by cybercrime, cyber espionage, and finally gray market business. Cybercrime is a very dangerous threat which consists of, for instance, stealing credentials of bank accounts, sending SMS to premium number, stealing user sensitive information, using resources of infected computer to develop e.g., spam business, DoS, botnets, etc. The interest of the cybercrime is to intentionally create malicious programs for its own interest, mostly lucrative. Hence, due to the malicious activity, cybercriminals have all the interest in not being detected during the attack, and developing their programs to be always more resilient against anti-malware solution. As a proof that this is a dangerous threat, the FBI reported a decline in physical crime and an increase of cybercrime [1].

In order to deal with the increasing number of exploits found in legacy code and to detect malicious code which leverages every subtle hardware and software detail to escape from malware analysis tools, the security research community started to develop and improve various code analysis techniques (static, dynamic or both), with the aim to detect the different forms of stealthy malware and to individuate security bugs in legacy code. Despite the improvement of the research solutions, yet the current ones are inadequate to face new stealthy and mobile malware.

Following such a line of research, in this dissertation[1], we present new program analysis techniques that aim to improve the analysis environment and deal with mobile malware.

To perform malware analysis, behavior analysis technique is the prominent: the actions that a program is performing during its real-time execution are collected to understand its behavior. Nevertheless, they suffer of some limitations.

State-of-the-Art malware analysis solutions rely on emulated execution environment to prevent the host to get infected, quickly recover to a pristine state, and easily collect process information. A drawback of these solutions is the non-transparency, that is, the execution environment does not faithfully emulate the physical end-user environment, which could lead to end up with incomplete results. In fact, malicious programs could detect when they are monitored in such environment, and thus modifying their behavior to mislead the analysis and avoid detection. On the contrary, a faithful emulator would drastically reduce the chance of detection of the analysis environment from the analyzed malware. To this end, we present EmuFuzzer, a novel testing methodology specific for CPU emulators, based on fuzzing to verify whether the CPU is properly emulated or not.

Another shortcoming regards the stimulation of the analyzed application. It is not uncommon that an application exhibit certain behaviors only when exercised with specific events (i.e., button click, insert text, socket connection, etc.). This flaw is even exacerbated when analyzing mobile application. At this aim, we introduce CopperDroid, a program analysis tool built on top of QEMU to automatically perform out-of-the-box dynamic behavior analysis of Android malware. To this end, CopperDroid presents a unified analysis to characterize low-level OS-specific and high-level Android-specific behaviors.

[1] All the technical work in this dissertation has been done before joining FireEye, Inc. and UC Berkeley.

Thanks for having believed in me

Contents

1 Introduction **1**
1.1 Dissertation Contributions . . . 4
1.2 Dissertation organization . . . 6

2 Architecture Preliminaries **7**
2.1 IA-32 Intel Architecture . . . 7
2.2 The ARM Architecture . . . 10

3 A methodology for testing CPU emulators **11**
3.1 Related Literature . . . 13
3.1.1 Software Testing . . . 13
3.1.2 Emulators and Computer Security . . . 14
3.2 Overview . . . 15
3.2.1 CPU Emulators . . . 15
3.2.2 Faithful CPU Emulation . . . 15
3.2.3 Fuzzing and Differential Testing of CPU Emulators . . . 16
3.3 EmuFuzzer . . . 18
3.3.1 Test Case Generation . . . 19
3.3.2 The Decoder . . . 23
3.3.3 Test Case Execution . . . 28
3.4 Evaluation . . . 32
3.4.1 A Glimpse at the Implementation . . . 33
3.4.2 Experimental Setup . . . 34
3.4.3 Evaluation of Test Case Generation . . . 34
3.4.4 Testing of IA-32 Emulators . . . 35

4 On Reconstructing Android Malware Behaviors 40
4.1 The Android System . . . 42
4.1.1 Application components . . . 43
4.1.2 Manifests . . . 44
4.1.3 Native Interface . . . 44
4.1.4 Zygote . . . 45
4.1.5 Binder: IPC and RPC . . . 45
4.2 Related Literature . . . 46
4.2.1 Current Techniques . . . 46
4.3 CopperDroid . . . 49
4.3.1 CopperDroid Architecture . . . 50
4.3.2 Processes and Threads . . . 51
4.3.3 Tracking System Call Invocations . . . 51
4.3.4 Automatic AIDL Unmarshalling . . . 52
4.3.5 Resource Reconstructor . . . 55
4.3.6 Path Coverage . . . 56
4.3.7 Suspicious Behaviors . . . 59
4.4 Evaluation . . . 61
4.4.1 Performance Evaluation . . . 63

5 On the Privacy of Real-World Friend-Finder Services 69
5.1 Background . . . 69
5.2 Attack description . . . 71
5.2.1 Scenario definition . . . 71
5.2.2 “Known distances” attack . . . 71
5.2.3 “Unknown distances” attack . . . 72
5.3 Attack automation . . . 73
5.3.1 Development of ad-hoc client . . . 74
5.3.2 Attack Algorithm . . . 75
5.4 Privacy Implications . . . 76
5.4.1 “Who is there?” attack . . . 76
5.4.2 “Where is Alice?” attack . . . 76
5.4.3 “Follow Alice” attack . . . 77
5.5 Ethical Considerations . . . 77
5.6 Conclusions . . . 78

6 Future directions 80
6.1 A methodology for testing CPU emulators . . . 80
6.2 On Reconstructing Android Malware Behaviors . . . 81

7 Conclusion 82
7.1 A methodology for testing CPU emulators 82
7.2 On Reconstructing Android Malware Behaviors 83

1 Introduction

With the term *malware*, or malicious software, it is identified any piece of code explicitly designed with the intent to cause damage to targets (i.e., users, companies or even authorities) and compromise their systems to perform frauds or espionage. Specifically, the NIST [2] defines it as:

> *"Malware, also known as malicious code and malicious software, refers to a program that is inserted into a system, usually covertly, with the intent of compromising the confidentiality, integrity, or availability of the victim's data, applications, or operating system or otherwise annoying or disrupting the victim."*

Malware have become the widespread and significant threat to most systems. Even thought they just born as computer vandalism, nowadays the main interest addresses the user's privacy violation. This risk, in fact, has become one of the major concern of companies and authorities as this form of malicious software monitors personal activities and conduct financial frauds. Even though for the last two decades the cybercrime mainly has targeted commodity PCs, with the advent and the steep increase of mobile devices, a new resource of interest for criminals comes to life. As depicted in Figure 1.1, the number of mobile threats impacting our daily life is skyrocketing. In fact, criminals realized that, thanks to their diffusion (750 million of activated android devices in 2013 [3]), mobile devices can turn into a remarkable resource of income by spreading mobile malware to perform any kind of illegal activity.

Mobile malware introduce new form of threats: *malware shopping spree* which make profit by buying applications on the store without the user permission; *NFC worms* which use the NFC capabilities to propagate and steal money; *SMS trojan* which fool the user into sending SMS to premium number; *Aggressive Advertising* that forces the redirection of the user to website with advertisement; *Spyware*

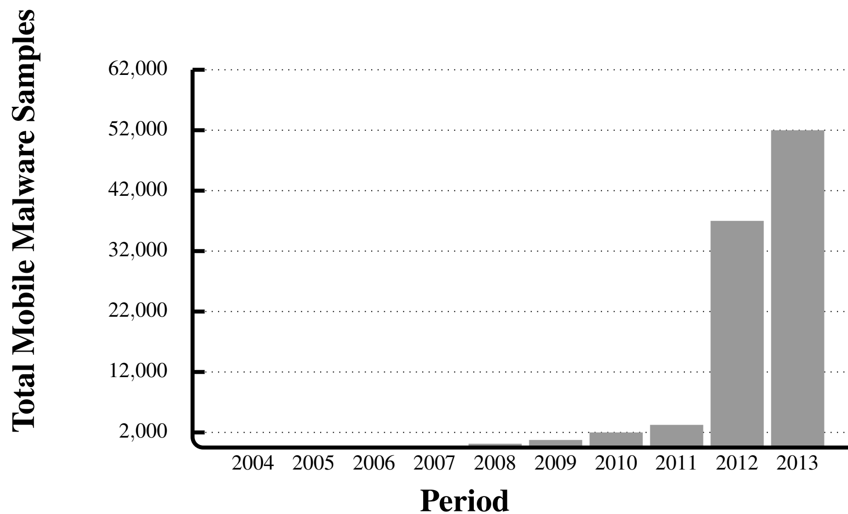


Figure 1.1: Mobile Threats (*source*: McAfee [4])

which steal personal and sensitive information, etc. This brief list shows that users don't have to drop their guard and the lucrative aspect of the malware dominates in the target of a criminal.

Another security aspect that is worth noting affects *BYOD* (Bring Your Own Device). Companies provide remote access to various services, including the critical ones, to their employees and partners to improve productivity and reduce the operating costs. As long as the IT maintained the control over the end-user devices, the security concerns were still negligible worries. However, in the last couple of years, companies have allowed user to bring and use their own *insecure* devices to get access to enterprise applications. This turned out to be a significant risk. Indeed, is fairly easy for a malware to steal user credentials, takeover the user enterprise account and eventually get access to the corporate sensitive information. This is even aggravate by the unawareness of the end-user about the security risks due to jailbreak a device, install third party apps, unpatch software, do not locking a device or using even benign applications that actually required a set of permissions that lead to sensitive information leakage. Moreover, due to lack of software update released by the vendor and, sometimes, the impossibility to wipe-out a device when is stolen or lost, the security threat becomes a very tough task to deal with.

Thus, the mobile world is not free of threats. On the contrary, it is getting even worse than the PCs world and performing detailed analysis of mobile applications became essential. The malicious software needs to be recognized as soon as it starts to spread to quickly develop new defence strategies. To this end, static and dynamic analysis techniques are employed.

Static analysis is the analysis of a program that is performed without executing it, but only reasoning on the binary code or source code if available [5, 6]. Unfortunately, the application of static analysis to malicious programs suffers of theoretical limitations that prevent precision of the overall results [7]. In fact, it can be easily fooled with encryption, polymorphism, metamorphism or different kind of code obfuscation techniques [8]. Dynamic analysis techniques come in handy to tackle these problems. These techniques should guarantee full code coverage, which means that every possible execution path of the analyzed program has to be observed. Nevertheless, this problem can be reduced to the halting problem and hence impossible to achieve. In fact, dynamic approaches can only reason on a limited number of program paths, i.e., the ones observed during the program execution. This leads to consider a malware a benign application if it does not exhibit its malicious behavior during the execution. For example, keylogger starts logging whenever a keyboard button is pressed or bank credentials are stolen if a user visit a specific bank website. This limitation forces the use of heuristics to improve code coverage, but, obviously, this does not come without any flaw (e.g., non negligible run-time overhead). State of the art solutions try to enhance heuristic approaches by exploring interesting paths, mostly leveraging taint-analysis and symbolic execution [9, 10]. Nevertheless, such information flow analyses techniques can be defeated by simple but powerful evasion techniques [11, 12]. Even with its shortcomings, dynamic analysis is actually the technique currently employed for pursuing malware behavior analysis [13, 14]. A suspicious program should be considered malicious if it exhibits a malicious behavior regardless of its binary representation. Generally, dynamic behavior analysis is performed in isolated execution environment to prevent the host to get infected, quickly recover to a pristine state, easily collect program information, and thereby safely analyze the application. This implies the need of an isolated execution environment which provides full-transparency and bulletproof separation between host and guest. In other words, a program running in this environment should not be able to infer that is not natively executed. This is a very hard task to achieve. Thus, by leveraging discrepancies between the emulated and native environment, authors of malware incorporate special pieces of code (*red-pills*) in their malicious programs to verify if they are executed in an emulated environment, and obfuscate their behavior if they suspect their execution is actually monitored.

Despite the improvement of the research solutions, yet the current ones are inadequate to face new stealthy mobile malware.

Following such a line of research, in this dissertation we present new program analysis techniques that aim to improve the analysis environment and deal with mobile malware.

1.1 Dissertation Contributions

As explained above, analysts employ CPU emulators as an execution environment to perform any kind of dynamic program analysis. A CPU emulator is a software system that simulates a hardware CPU. Emulators are widely used by computer scientists for various kind of activities (e.g., debugging, profiling, and malware analysis). Although no theoretical limitation prevents developing an emulator that faithfully emulates a physical CPU, writing a fully featured emulator is a very challenging and error prone task. Modern CISC architectures have a very rich instruction set. Some instructions lack proper specifications, and others may have undefined effects in corner cases. In the first part of this dissertation we present a testing methodology specific for CPU emulators, based on fuzzing. The emulator is "stressed" with specially crafted test cases, to verify whether the CPU is properly emulated or not. Improper behaviors of the emulator are detected by running the same test case concurrently on the emulated and on the physical CPUs and by comparing the state of the two after the execution. Differences in the final state testify defects in the code of the emulator. We implemented this methodology in a prototype (named as EmuFuzzer), analyzed five state-of-the-art IA-32 emulators (QEMU, Valgrind, Pin, BOCHS, and JPC), and found several defects in each of them, some of which can prevent proper execution of programs.

To further support and motivate the importance of this technique, we can consider that mobile devices that boast of thousands of applications in their respective vendor markets, require the developers to rely on emulators to test their applications during the software development life-cycle.

Besides this novel testing methodology, which basically addresses the execution environment, new program analysis technique are required to analyze mobile applications. Specifically, with more than 500 million of activations reported in Q3 2012, Android mobile devices are becoming ubiquitous and trends confirm this is unlikely to slow down. App stores, such as Google Play, drive the entire economy of mobile applications. Unfortunately, high turnovers and access to sensitive data have soon attracted the interests of cybercriminals with malware now hitting Android devices at an alarming rising pace. In the second part of this dissertation we present CopperDroid, an approach built on top of QEMU to automatically perform out-of-the-box dynamic behavioral analysis of Android malware. To this end, CopperDroid presents a unified analysis to characterize low-level OS-specific and high-level Android-specific behaviors. Based on the observation that such behaviors are however achieved through the invocation of system calls, CopperDroid's VM-based dynamic system call-centric analysis is able to faithfully describe the behavior of Android malware whether it is initiated from Java, JNI or native code execution. We carried out extensive experiments to assess the effectiveness of our analyses on three different Android malware data set: one of more

than 1,200 samples belonging to 49 Android malware families (Android Malware Genome Project), one containing about 400 samples over 13 families (Contagio project) and a last one, previously unanalyzed, made of more than 1,300 samples, provided by McAfee. Our experiments show that CopperDroid's unified system call-based analysis faithfully describes OS- and Android-specific behaviors and a proper malware stimulation strategy (e.g., sending SMS, placing calls) successfully discloses additional behaviors on a non-negligible portion of the analyzed malware samples.

CopperDroid does not just address analysis of malicious programs, but also allows to perform a deep and detailed analysis of every application. To stress the advantages of such a solution, we present the analysis of a location aware mobile application as a case-study. We show that even benign applications can lead to privacy leakage when the involved sensitive information are not subjected to any sort of protection to provide privacy data retention. This is mainly due to the developer awareness and consideration of possible attacks. Privacy protection in the deployment of location based services is a hot topic both in CS research and in the development of mobile applications. We consider a location based service that currently has hundreds of millions of users and we show a software that is able to discover their exact positions, by only using information publicly disclosed by the service. Our software does not exploit a specific limitation of the considered service. Rather this contribution shows that there is an entire class of services that is subject to the attack we present.

This dissertation presents novel solutions that aim to provide new approaches and overcome the shortcomings as well as enhance and improve current dynamic program analysis techniques. To summarize, we make the following contributions:

A methodology for testing CPU emulators. Lorenzo Martignoni, Roberto Paleari, Alessandro Reina, Giampaolo Fresi Roglia, Danilo Bruschi. *ACM Transactions on Software Engineering and Methodology 2013 (TOSEM 2013)*

A System Call-Centric Analysis and Stimulation Technique to Automatically Reconstruct the Behaviors of Android Malware. Alessandro Reina, Aristide Fattori, Lorenzo Cavallaro. 6th *European Workshop on Systems Security (EUROSEC 2013)*

Automatic Reconstruction of Android Malware Behaviors. Kimberly Tam, Alessandro Reina, Aristide Fattori, Lorenzo Cavallaro. 18th *European Symposium on Research in Computer Security. (Abstract - ESORICS 2013)*

On the Privacy of Real-World Friend-Finder Services. Aristide Fattori, Alessandro Reina, Andrea Gerino, Sergio Mascetti. *14th International Conference on Mobile Data Management (MDM 2013)*

1.2 Dissertation organization

The dissertation is organized as follows.

Chapter 2 briefly reviews the main fundamental features of the Intel IA-32 and ARM architectures.

Chapter 3 presents EmuFuzzer, a novel testing methodology based on fuzzing specific for CPU emulators. We describe our algorithms for test-case generation and how test cases are run to detect if an emulator is not faithfully emulating the CPU. We evaluate our methodology by presenting the results of the testing of five CPU emulators.

Chapter 4 introduces CopperDroid, a program analysis tool build on the top of QEMU to automatically perform out-of-the-box dynamic behavior analysis of Android malware. We describe our stimulation technique to perform path coverage and we experimentally evaluate our solution.

Chapter 5 presents a use-case of CopperDroid which is employed to analyze a benign application that actually threatens the user-privacy.

Chapter 6 discusses limitations and future works.

Chapter 7 concludes the dissertation.

2 Architecture Preliminaries

The program analysis solutions discussed and explained in this dissertation, even though closely related in their aim, concern different architectures. To this end, we briefly review the background of IA-32 and ARM architectures necessary to understand the following chapters.

2.1 IA-32 Intel Architecture

The IA-32 refers to a family of 32-bit Intel processors that are widely used in many multi-purpose environments because of their facilities and performance. In this section we provide a brief introduction to the IA-32 architecture. For further details, an interested reader can refer elsewhere [15].

IA-32 is a CISC architecture, with an incredible number of different instructions and a complex encoding scheme. Instruction length can vary from 1 to 17 bytes. The format of an Intel x86 instruction is depicted in Figure 2.1. An instruction is composed of different fields: it starts with up to 4 prefixes, followed by an opcode, an addressing specifier (i.e., `ModR/M` and `SIB` fields), a displacement and an immediate data field [15]. Opcodes are encoded with one, two, or three bytes, but three extra bits of the `ModR/M` field can be used to denote certain opcodes. In total, the instruction set is composed of more than 700 possible values of the opcode field. The `ModR/M` field is used in many instructions to specify non-implicit operands: the `Mod` and `R/M` sub-fields are used in combination to specify either registry operands or to encode addressing modes, while the `Reg/Opcode` sub-field can either specify a register number or, as mentioned before, additional bits of opcode information. The `SIB` byte is used with certain configurations of the `ModR/M` field, to specify base-plus-index or scale-plus-index addressing forms. The `SIB` field is in turn partitioned in three sub-fields: `Scale`, `Index`, and `Base`, speci-

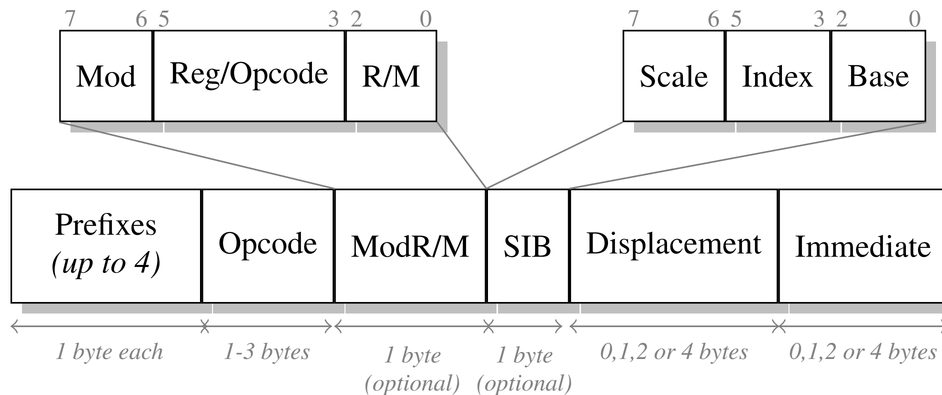


Figure 2.1: Intel x86 instruction format

fying respectively the scale factor, the index register, and the base register. Finally, the optional addressing displacement and immediate operands are encoded in the `Displacement` and `Immediate` fields respectively. Since the encoding of the `ModR/M` and `SIB` bytes is not trivial at all, the Intel x86 specification provides tables describing the semantics of the 256 possible values each of these two bytes might assume. In conclusion, it is easy to see that elementary decoding operations, such as determining the length of an instruction, require decoding the entire instruction format and interpreting the various fields correctly. In recent years, the advent of several instruction extensions (e.g., Multiple Math eXtension (MMX) and Streaming SIMD Extensions (SSE)) contributed to make the instruction set even more complicated.

The IA-32 architecture supports four basic operating modes: real-address mode, protected mode, virtual-8086 mode, and system management mode. The operating mode of the processor determines which instructions and architectural features are available. Every operating mode implies a well-defined set of instructions and semantics, and some instructions behave differently depending on the mode. For example, instruction can raise different exceptions and can update flags and registers differently when executed in the protected mode and when executed in the virtual-8086 mode.

Any task or program running on an IA-32 processor is given a set of resources for storing code, data, state information, and for executing instructions. These resources constitute the basic execution environment and they are used by both the operating system and users' applications. The resources of the basic execution environment are identified as follows:

- **Address space**: any task or program can address a 32-bit linear address space;

- **Basic program execution environment**: the eight general-purpose registers (`eax`, `ecx`, `edx`, `ebx`, `esp`, `ebp`, `esi`, `edi`), the six segment registers (`cs`, `ss`, `ds`, `es`, `fs`, `gs`), the `eflags` register, and the `eip` register comprise a basic execution environment in which to execute a set of general-purpose instructions;

- **Stack**: to support procedure or subroutine calls and the passing of parameters between procedure and subroutines;

- **x87 FPU registers**: this set of registers provides an execution environment for floating point operations;

- **MMX registers** and **XMM registers**: registers used by dedicated instructions designed for accelerating multimedia and communication applications.

In addition to these resources, the IA-32 architecture provides the following resources as part of its system-level architecture.

- **I/O ports**: the IA-32 architecture supports a transfer of data to and from input/output ports;

- **Control register**: the five control registers (`cr0` through `cr4`) determine the operating mode of the processor and the characteristics of the currently executing task;

- **Memory management register**: the `gdtr`, `idtr`, task register, and `ldtr` specify the locations of data structures used in protected mode memory management;

- **Debug register**: the debug registers (`db0` through `db7`) control and allow monitoring of the processor's debugging operations;

- **Memory type range registers**: the memory type range registers are used to assign memory type to regions of memory such as: uncacheable, write combining, write through, write back, and write protected type;

- **Machine specific registers**: the processor provides a variety of machine specific registers (MSR) that are used to control and report on processor performance;

- **Machine check registers**: the machine check registers consist of a set of control, status, and error-reporting MSRs that are used to detect and report on hardware (machine) errors. Specifically the IA-32 processors implement a machine check architecture that provides a mechanism for detecting and reporting errors such as: system bus errors, ECC errors, parity errors, cache errors, and TLB errors.

CPU emulators have to offer an execution environment suitable for running an application or even a commodity operating system. Given the complexity of IA-32 architecture, fully featured CPU emulators for this architecture are complex pieces of software. Our claim is that this complexity is the cause of a large number of defects.

2.2 The ARM Architecture

ARM processors [16] are the de-facto standard commodity CPUs for embedded systems, mostly because of their appealing features: low-power consumptions, high-code density, performance, small chip size and low-cost solutions. ARM is a 32-bit load-store architecture with 4-bytes instruction length and 18 active registers (i.e., 16 data registers and 2 processor status registers). ARM is not a pure RISC architecture because of the constraints of its application. In addition to RISC, it provides variable cycle execution for certain instructions (e.g., load-store instructions cycles depend on the number of registers involved), inline hardware barrel shifter to expand capability of many instructions, thumb 16-bit instruction set to increase code density, conditional execution to reduce branch instructions and DSP instructions. ARM general purpose registers, identified with `r` followed by the number of the registers, hold either data or address. Special-purpose registers, `r13`, `r14` and `r15`, are designed to respectively represent the stack pointer (`sp`), the link register (`lr`) that contains the return address and the program counter (`pc`). The current program status register, `cpsr`, is a 32-bit register designed to monitor and control internal operations: flags, status, extension and control. The processor mode, whose value is contained in the `cpsr`, is the equivalent of the privilege level of Intel x86 and amd64 architectures and determines which register are active and the access rights to the `cpsr` itself. Each of the seven processor modes is either *privileged* or *non-privileged*. The former allows full read-write access to the `cpsr` register while the latter allows read access to the control field of the `cpsr` and read-write to the conditional flags. Each processor mode has its own `banked registers` (i.e., a subset of the active registers) the are replaced with the current ones when happens a mode change. Specifically, there is one non-privileged mode, `user`, and six privileged modes `abort`, `fast interrupt request`, `interrupt request`, `supervisor`, `system` and `undefined` [1].

[1] For sake of simplicity, you can consider Intel `ring3` privilege level as the ARM `user` processor mode, and Intel `ring0` privilege level as the ARM `supervisor` processor mode.

3 A methodology for testing CPU emulators

In Computer Science, the term "*emulator*" is typically used to denote a piece of software that simulates a hardware system [17]. Different hardware systems can be simulated: a device [18], a CPU (Pin [19] and Valgrind [20]), and even an entire PC system (QEMU [21], BOCHS [22], JPC [23], and Simics [24]). Emulators are widely used today for many applications: development, debugging, profiling, security analysis, *etc*. For example, the NetBSD AMD64 port was initially developed using an emulator [25].

The Church-Turing thesis implies that any effective computational method can be emulated within any other. Consequently, any hardware system can be emulated via a program written with a standard programming language. Despite the absence of any theoretical limitation that prevents the development of a correct and complete emulator, from the practical point of view, the development of such a software is very challenging. This is particularly true for *CPU emulators*, that simulate a physical CPU. Indeed, the instruction set of a modern CISC CPU is very rich and complex. Moreover, the official documentation of CPUs often lacks the description of the semantics of certain instructions in certain corner cases and sometimes contains inaccuracies (or ambiguities). Although several good tools and debugging techniques exist [26], developers of CPU emulators have no specific technique that can help them to verify whether their software emulates the CPU by following precisely the specification of the vendors. As CPU emulators are employed for a large variety of applications, defects in their code might have cascading implications. Imagine, for example, what consequences the existence of any defect in the emulator used for porting NetBSD to AMD64 would have had on the reliability of the final product.

Assuming that the physical CPU is correct by definition, the ideal CPU emulator has to mimic exactly the behavior of the physical CPU it is emulating. On the contrary, an approximate emulator deviates, in certain situations, from the behav-

ior of the physical CPU. There are particular examples of approximate emulators in literature [27–31]. Our goal is to develop a general automatic technique to discover deviations between the behavior of an emulator and of the corresponding physical CPU. In particular, we are interested in investigating deviations (i.e., state of the CPU registers and contents of the memory) which could modify the behavior of a program in an emulated environment. On the other hand, we are not interested in deviations that lead only to internal differences in the state (e.g., differences in the state of CPU caches), because these differences do not affect the behavior of the programs running inside the emulated environment.

In this dissertation we present a fully automated and black-box testing methodology for CPU emulators, based on fuzzing [32]. Roughly speaking such a methodology works as follows. Initially we automatically generate a very large number of test cases. Strictly speaking, a test case is a single CPU instruction together with an initial environment configuration (CPU registers and memory contents); a more formal definition of a test case is given in section 3.2.3. These test cases are subsequently executed both on the physical CPU and on the emulated CPU. Any difference detected in the configurations of the two environments (e.g., register values or memory contents) at the end of the execution of a test case, is considered a witness of an incorrect behavior of the emulator. Given the unmanageable size of the test case space, we adopt two strategies for generating test cases: purely random test case generation and hybrid algorithmic/random test case generation. The latter guarantees that each instruction in the instruction set is tested at least in some selected execution contexts. We have implemented this testing methodology in a prototype for IA-32, named as EmuFuzzer, and used it to test five state-of-the-art emulators: BOCHS [22], QEMU [21], Pin [19], Valgrind [20], and JPC [23]. Although Pin and Valgrind are dynamic instrumentation tools, their internal architecture resembles, in all details, the architecture of traditional emulators and therefore they can suffer from the same problems. We found several deviations in the behaviors of each of the five emulators. Some examples of the deviations we found in these state-of-the-art emulators are reported in Table 3.1[1]. As an example, let us consider the instruction `add $0x1,(%eax)`, which adds the immediate 0x1 to the byte pointed by the register `eax`. Assuming that the original value of the byte is `0xcf`, the execution of the instruction on the physical CPU, and on four of the tested emulators, provides the result `0xd0`. In QEMU, instead, the value is not updated correctly for a certain encoding of the instruction. We also discovered instructions that are correctly executed in the native environment but freeze QEMU and instructions that are not supported by Valgrind and thus generate exceptions. On the other hand we also found instructions that are executed by Pin and BOCHS but that cause exceptions on the physical CPU. The results obtained witness the

[1] In this dissertation we use IA-32 assembly and we adopt the AT&T syntax.

Table 3.1: Examples of instructions that behave differently when executed in the physical CPU and when executed in an emulated CPU (that emulates an IA-32 CPU). For each instruction, we report the behavior of the physical CPU and the behavior of the emulators (differences are highlighted)

Instruction	IA-32	QEMU	Valgrind	Pin	BOCHS	JPC
`lock fcos`	illegal instr.	`lock` ignored	*no diff.*	*no diff.*	*no diff.*	`lock` ignored
`int1`	trap	*no diff.*	illegal instr.	*no diff.*	general prot. fault	not supported
`fld1`	`fpuip` = `eip`	`fpuip` = 0	`fpuip` = 0	FPU virtualized[2]	*no diff.*	`fpuip` = 0
`add $0x1,(%eax)`	`(%eax) = 0xd0`	`(%eax) = 0xcf`	*no diff.*	*no diff.*	*no diff.*	*no diff.*
`pop %fs`	`%esp = 0xbfdbb108`	*no diff.*	*no diff.*	`%esp = 0xbfdbb106`	*no diff.*	segment not present
`pop 0xffffffff`	`%esp = 0xbffffe44`	*no diff.*	*no diff.*	*no diff.*	`%esp = 0xbffffe48`	*no diff.*

difficulty of writing a fully featured and specification-compliant CPU emulator, but also prove the effectiveness and importance of our testing methodology.

The main contributions of this work are as follows:

- a fully automated testing methodology, based on fuzz-testing, specific for CPU emulators;
- an optimized algorithm for test case generation that systematically explores the instruction set, while minimizing redundancy;
- a prototype implementation of our testing methodology for IA-32 emulators;
- an extensive testing of five IA-32 emulators that resulted in the discovery of several defects in each of them, some of which represent serious bugs.

3.1 Related Literature

3.1.1 Software Testing

Fuzz-testing has been introduced by Miller *et al.* [32], and it is still widely used for testing different types of applications. Originally, fuzz-testing consisted of feeding applications purely random input data and detecting which inputs were able to crash an application, or to cause unexpected behaviors. Today, this testing methodology is used to test many different types of applications; for example, GUI applications, web applications, scripts, and kernel drivers [33].

As certain applications require inputs with particular format (e.g., a XML document or a well formed Java program), pure randomly generated inputs cannot

[2] PIN virtualizes the physical FPU, so floating point instructions are executed natively rather than being emulated.

guarantee a reasonable coverage of the code of the application under analysis. Recently developed testing techniques typically leverage domain specific knowledge and use this knowledge, optionally in tandem with a random component, to drive inputs generation [34–36]. An alternative approach to improve the completeness of the testing consists of building constraints that describe what properties are required for the input to trigger the execution of particular program paths, and in using a constraint solver to find inputs with these properties [37–42]. In this dissertation we presents a fuzz-testing methodology specific for CPU emulators that leverages both pure random inputs generation and domain knowledge to improve the completeness of the analysis.

In our previous works, we explored the idea of using mechanically generated tests and to compare the behavior of two components to detect deviations imputable to bugs [43–45]. This approach is known in literature as differential testing [46–49]. EmuFuzzer adopts differential testing to detect if the tested CPU emulator behaves unfaithfully with respect to the physical CPU emulated.

3.1.2 Emulators and Computer Security

CPU emulators are widely used in computer security for various purposes. One of the most common applications is malware analysis [14, 50]. Emulators allow fine-grained monitoring of the execution of a suspicious programs and to infer high-level behaviors. Furthermore they allow to isolate the execution and to easily checkpoint and restore the state of the environment. Malware authors, aware of the techniques used to analyze malware, aim at defeating those techniques such that their software can survive longer. To defeat dynamic behavioral analysis based on emulators, they typically introduce malware routines able to detect if a program is executed in an emulated or in a physical environment. As the average user targeted by the malware does not use emulators, the presence of an emulated environment likely indicates that the program is being analyzed. Thus, if the malicious program detects the presence of an emulator, it starts to behave innocuously such that the analysis does not detect any malicious behavior. Several researchers have analyzed state-of-the-art emulators to find unfaithful behaviors that could be used to write specific detection routines [28, 30, 31, 51]. Unfortunately for them, their results were obtained through a manual scrutiny of the source code or rudimentary fuzzers, and thus the results are largely incomplete. The testing technique presented in this dissertation can be used to find automatically a large class of the unfaithful behaviors that a miscreant could use to detect the presence of an emulated CPU. This information could then be used to harden an emulator, to the point that it satisfies the requirements for undetectability identified by Dinaburg *et al.* [52].

3.2 Overview

This section describes how CPU emulators work, formalizes our notion of faithful emulation of a physical CPU, and sketches the idea behind our testing methodology.

3.2.1 CPU Emulators

By CPU emulator we mean a piece of software system that simulates the execution environment offered by a physical CPU. The execution of a binary program P is emulated when each instruction of P is executed by a CPU emulator. Inside a CPU emulator instructions are typically executed using either interpretation or just-in-time translation. Here, we are only interested in emulators adopting the former strategy, in such case instructions are executed by mimicking in every detail the behavior of the physical CPU, obviously operating on the resources of the emulated execution environment.

The execution environment can be properly emulated even if some internal components of the physical CPU are not considered (e.g., the instruction cache): as these components are used transparently by the physical CPU, no program can access them. Similarly, emulated execution environments can contain extra, but transparent, components not found in hardware execution environments (e.g., the cache used to store translated code).

3.2.2 Faithful CPU Emulation

Given a physical CPU $\mathscr{C}_P$, we denote with $\mathscr{C}_E$ a software CPU emulator that emulates $\mathscr{C}_P$. Our ideal goal is to automatically analyze a given $\mathscr{C}_E$ to tell whether it *faithfully emulates* $\mathscr{C}_P$. In other words we would like to tell if $\mathscr{C}_E$ behaves equivalently to $\mathscr{C}_P$, in the sense that any attempt to execute a valid (or invalid) instruction results in the same behavior in both $\mathscr{C}_P$ and $\mathscr{C}_E$. In the following we introduce some definitions which will help us to precisely define this equivalence notion.

Let N be the number of bits used by a CPU $\mathscr{C}$ for representing its memory addresses as well as the registers contents. A state s of $\mathscr{C}$ is represented by the following tuple $s = (pc, R, M, E)$ where

- $pc \in \{0, \ldots, 2^N - 1\} \cup \texttt{halt}$;
- $R = < r_1, \ldots, r_k >$; $r_i \in \{0, \ldots, 2^N - 1\}$ is the value contained in the ith CPU register;

- $M = < b_0, \ldots, b_{2^N-1} >$; $b_i \in \{0, \ldots, 255\}$ is the contents of the ith memory byte;
- $E \in \{\perp, \textit{illegal instruction}, \textit{division by zero}, \textit{general protection fault}, \ldots\}$ denotes the exception that occurred during the execution of the last instruction; the special exception state $\perp$ indicates that no exception occurred.

We denote by $\mathscr{S}$ the set of all states of a CPU. The behavior of a CPU $\mathscr{C}$ is modeled by a transition system $(\mathscr{S}, \delta_{\mathscr{C}})$, where $\delta_{\mathscr{C}} : \mathscr{S} \rightarrow \mathscr{S}$ is the state-transition function which maps a CPU state $s = (pc, R, M, E)$ into a new state $s' = (pc', R', M', E')$ by executing the instruction whose address is specified by the pc. The transition function δ is defined as follows:

$$\delta_{\mathscr{C}}(pc, R, M, E) \stackrel{\text{def}}{=} \begin{cases} (pc, R, M, E) & \text{if } pc = \texttt{halt} \vee E \neq \perp, \\ (pc, R, M, E') & \text{if an exception occurs,} \\ (pc', R', M', \perp) & \text{otherwise.} \end{cases}$$

When $E' \neq \perp$ the contents of the registers R', of the memory M' and of pc' are updated according to the semantics of the executed instruction. On the other side, if an exception occurs, then we assume for simplicity[3] that $\delta_{\mathscr{C}}(pc, R, M, E) = (pc, R, M, E')$. When the last instruction of a program is executed, the program counter is set to `halt`, and from that point on the state of the environment is not updated anymore.

We can now formally define what it means for $\mathscr{C}_E$ to be a *faithful emulator* of $\mathscr{C}_P$. Intuitively, $\mathscr{C}_E$ faithfully emulates $\mathscr{C}_P$ if the state-transition function $\delta_{\mathscr{C}_E}$ that models $\mathscr{C}_E$ is semantically equivalent to the function $\delta_{\mathscr{C}_P}$ that models $\mathscr{C}_P$. That is, for each possible state $s \in \mathscr{S}$, $\delta_{\mathscr{C}_P}$ and $\delta_{\mathscr{C}_E}$ always transition into the same state. More formally, $\mathscr{C}_E$ *faithfully emulates* $\mathscr{C}_P$ iff:

$$\forall s \in \mathscr{S} : \delta_{\mathscr{C}_P}(s) = \delta_{\mathscr{C}_E}(s).$$

3.2.3 Fuzzing and Differential Testing of CPU Emulators

Given a physical CPU $\mathscr{C}_P$ and an emulator $\mathscr{C}_E$, proving that $\mathscr{C}_E$ faithfully emulates $\mathscr{C}_P$ is unfeasible as it requires the verification of a huge number of states. Thus, our aim is to find witnesses of the fact that an emulator $\mathscr{C}_E$ does not faithfully emulate $\mathscr{C}_P$.

We achieve this goal by generating a number of test cases, i.e., CPU states $s = (pc, R, M, E)$, and looking for a test case $\bar{s}$ which proves that $\mathscr{C}_E$ *unfaithfully*

[3]Exceptions actually modify CPU registers and memory. However, in our model, when an exception occurs execution is interrupted, so these modifications can be safely ignored.

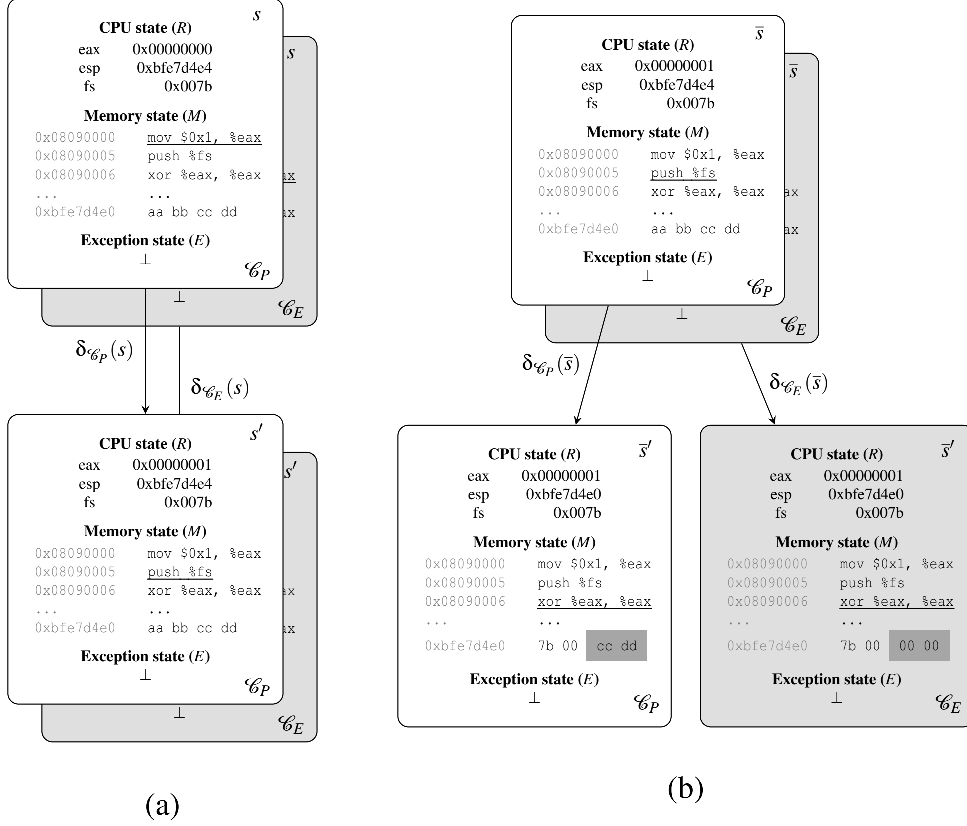


Figure 3.1: An example of our testing methodology with two different test cases (s and $\bar{s}$): (a) no deviation in the behavior is observed, (b) the words at the top of the stack differ (highlighted in gray).

emulates $\mathscr{C}_P$ i.e.[4]:

$$\bar{s} \in \mathscr{S} : \delta_{\mathscr{C}_P}(\bar{s}) \neq \delta_{\mathscr{C}_E}(\bar{s}).$$

Our approach for finding $\bar{s}$ is based on fuzzing [32] (for test case generation) and differential testing [46] (to compare $\delta_{\mathscr{C}_P}(s)$ against $\delta_{\mathscr{C}_E}(s)$). Once a test case s has been generated we set the state of both $\mathscr{C}_P$ and $\mathscr{C}_E$ to s. Then we execute the instruction pointed by pc in both $\mathscr{C}_P$ and $\mathscr{C}_E$. At the end of the execution of the instruction, we compare the final state. If no difference is found, then $\delta_{\mathscr{C}_P}(s) = \delta_{\mathscr{C}_E}(s)$ holds. On the other hand, a difference in the final state proves that $\delta_{\mathscr{C}_P}(s) \neq \delta_{\mathscr{C}_E}(s)$ and therefore that $\mathscr{C}_E$ does not faithfully emulate $\mathscr{C}_P$.

[4]Here we assume that δ is a function (hence deterministic) for a specific CPU model. Indeed, even if for some instructions the CPU specifications are not completely defined, it turns out that, given an initial state, the behavior of any instruction is deterministic. Obviously, CPU undefined behaviors are not documented in the released specifications, therefore emulators do not simulate them.

Figure 3.1 shows an example of our testing methodology[5]. We run two different test cases, namely s and $\overline{s}$. To ease the presentation, in the figure we report only the relevant state information (three registers and the contents of few memory locations) and we represent the program counter by underlining the instruction it is pointing to. Furthermore, when the states of the two environments do not differ, we graphically overlap them. The first test case s (Figure 3.1(a)) consists of executing the instruction `mov $0x1, %eax`. We set the state of $\mathscr{C}_P$ and $\mathscr{C}_E$ to s and we execute in both the instruction pointed by the program counter. As there is no difference in the final states, we conclude that $\delta_{\mathscr{C}_E}(s) = \delta_{\mathscr{C}_P}(s)$. The second test case $\overline{s}$ (Figure 3.1(b)) consists of executing the instruction `push %fs`, that saves the segment register `fs` on the stack. Although the register is 16 bits wide, the IA-32 specification dictates that, when operating in 32-bit mode, the CPU has to reserve 32 bits of the stack for the store. In the example we observe that $\mathscr{C}_P$ leaves the upper 16 bits of the stack untouched, while $\mathscr{C}_E$ overwrites them with zero (the different bytes are highlighted in the figure). The two final states differ because the contents of their memory differs, consequently, $\delta_{\mathscr{C}_P}(\overline{s}) \neq \delta_{\mathscr{C}_E}(\overline{s})$. That proves that $\mathscr{C}_E$ does not faithfully emulate $\mathscr{C}_P$.

3.3 EmuFuzzer

The development of the approach briefly described in the previous section requires overcoming two major difficulties. First, as the potential number of states in which an emulator should be tested is prohibitively large, we have to focus our efforts on selecting a small subset of states, which maximizes the completeness of the testing. Second, the detection of deviations in the behaviors of the two environments requires us to properly setup and inspect their state at the end of the execution of each test case. Thus, we need to develop a mechanism to efficiently initialize and compare the state of the two environments. In this section we provide a detailed description of how these difficulties have been overcome.

Although the methodology we are proposing is architecture independent, our implementation, called EmuFuzzer, is currently specific for IA-32. This choice is solely motivated by our limited hardware availability. Nevertheless, minor changes to the implementation would be sufficient to port it to different architectures. To ease the development, the current version of the prototype runs entirely in user-space and thus can only verify the correctness of the emulation of unprivileged instructions and whether privileged instructions are correctly prohibited. EmuFuzzer deals with two different types of emulators: process emulators that emulate a single process at a time (e.g., Valgrind, PIN, and QEMU), and

[5] This example reflects a real defect we have found in QEMU using our testing methodology.

whole-system emulators that emulate an entire system (e.g., BOCHS, JPC, and QEMU[6]).

3.3.1 Test Case Generation

As just mentioned, in our testing methodology, a test case $s = (pc, R, M, \perp)$ is a state of the environment under test. The memory contains the code that will be executed by the CPU, as well as the corresponding data part of which is contained in R. To generate test cases we adopt two strategies: (i) *random test case generation*, where both data and code are random, and (ii) *CPU-assisted test case generation*, where data is random, and code is generated algorithmically, with the support of the physical and of the emulated CPUs. The advantage of using two different strategies is a better coverage of the test case space. Test cases are generated by an assembly program, which contains instructions for environment initialization, i.e., memory and registers, and loads into the test case memory one single instruction, i.e., the instruction we want to test. Figure 3.2 shows a C pseudocode of such a program. This program initializes the state of the environment, by loading the memory content (lines 6–10) and the data in the CPU registers (lines 12–15), and subsequently it triggers the execution of the code of the test case (line 19). The program is compiled with appropriate compiler flags to generate a tiny self-contained executable (i.e., that does not use any shared library).

There are other possible approaches to generate the code of test cases. For example, one can generate assembly instructions and then compile them with an assembler or use a disassembler to detect which sequences of bytes encode a legal instruction. However, limitations of the assembler or of the disassembler negatively impact on the completeness of the generated test cases. Besides our approach, detailed in the following, none of the ones just mentioned can guarantee no false-negative (i.e., that a sequence of bytes encoding a valid instruction is considered invalid).

3.3.1.1 Random Test Case Generation

In random test case generation, both data and code of the test case are generated randomly. The memory is initialized by mapping a file filled with random data. For simplicity, the same file is mapped multiple times at consecutive addresses until the entire user-portion of the address space is allocated. To avoid a useless waste of memory, the file is lazily mapped in memory, such that physical memory pages are allocated only if they are accessed. The CPU registers are also initialized with random values. As we work in user-space, we cannot allocate the entire

[6]QEMU supports both whole-system and process emulation.

```
1 void main() {
2   void *p;
3   // Code of the test case
4   char code[] = "\xB8\xEF\xBE\xAD\xDE";
5
6   // Initialize the memory with random data
7   for (p = 0x0; p < FILE_SIZE; p += PAGE_SIZE) {
8     f = open(FILE_WITH_RANDOM_DATA, O_RDWR);
9     mmap(p, PAGE_SIZE, ..., MAP_FIXED, f, 0);
10  }
11
12  // Initialize the registers with random data
13  asm("mov RANDOM, %eax");
14  asm("mov RANDOM, %ebx");
15  asm("mov RANDOM, %ecx");
16  ...
17
18  // Execute the code of the test case (pc = code)
19  ((void(*)()) code)();
20 }
```

Figure 3.2: Pseudocode of the program which generates a test case.

address space because a part of it is reserved for the kernel. Therefore, to minimize page faults when registers are used to dereference memory locations, we make sure the value of general purpose registers fall around the middle of the allocated user address space. The rationale is to maximize the probability that, for any instruction, memory operands refer to valid locations. Obviously, code generated with this random approach might contain more than one instruction.

3.3.1.2 CPU-assisted Test Case Generation

A thorough testing of an emulator requires us to verify that each possible instruction is emulated faithfully. Unfortunately, the pure random test case generation approach presented earlier is very unlikely to cover the entire instruction set of the architecture (the majority of CPU instructions require operands encoded using specific encoding and others have opcodes of multiple bytes). Ideally, we would have to enumerate and test all possible instances of instructions (i.e., combinations of opcodes and operands). Clearly this is not feasible. To narrow the problem space, we identify all supported instructions and then we test the emulator using only few peculiar instances of each instruction. That is, for each opcode we generate test cases by combining the opcodes with some predefined operand

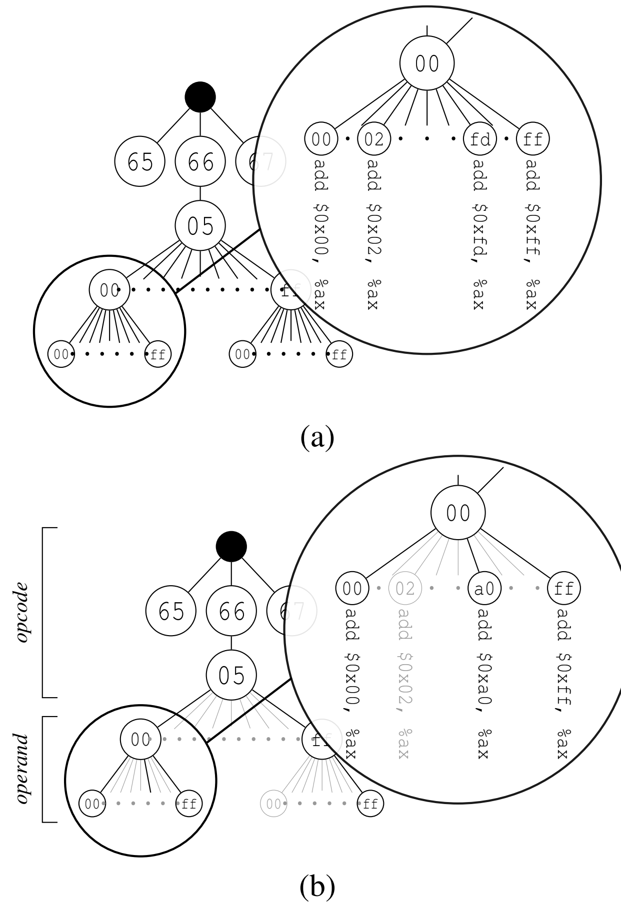


Figure 3.3: Example of CPU-assisted test case generation for the opcode `6605` (`mov imm16,%ax`): (a) naïve and (b) optimized generation (paths in gray are not explored).

values. As in random-test case generation, the data of the test case are random.

Naïve Exploration of the Instruction Set Our algorithm for generating the code of a test case leverages both the physical and the emulated CPUs, in order to identify byte sequences representing valid instructions. We call our algorithm *CPU-assisted test case generation*. The algorithm enumerates the sequences of bytes and discards all the sequences that do not represent valid code. The CPU is the oracle that tells us if a sequence of bytes encodes a valid instruction or not: sequences that raise illegal instruction exceptions do not represent valid code. We run our algorithm on the physical and on the emulated CPUs and then we take the union of the two sets of valid instructions found. The sequences of bytes that cannot be executed on both CPUs are discarded because they do not represent in-

teresting test cases: we know in advance that the CPUs will behave equivalently (i.e., $E' =$ *illegal instruction*). On the other hand, a sequence of bytes that can be executed on at least one of the two CPUs is considered interesting because it can lead to one of the following situations: (i) it represents a valid instruction for one CPU and an invalid instruction for the other; (ii) it encodes a valid instruction for both CPUs but, once executed, causes the CPUs to transition to two different states.

Optimized Exploration of the Instruction Set We can imagine representing all valid CPU instructions as a tree, where the root is the empty sequence of bytes and the nodes on the path from the root to the leaves represent the various bytes that compose the instruction. Figure 3.3(a) shows an example of such a tree. Our algorithm exploits a particular property of this tree in order to optimize the traversal and to avoid the generation of redundant test cases: the majority of instructions have one or more operands and thus multiple sequences of bytes, sharing the same prefix, encode the same instruction, but with different operands. In the following we describe an example of the optimized instruction set exploration; further details are then given in Section 3.3.2.

As an example, let us consider the 2^{16} sequences of bytes from `66 05 00 00` to `66 05 FF FF` that represent the same instruction, `add imm16,%ax`, with just different values of the 16-bit immediate operand. Figure 3.3(a) shows the tree representation of the bytes that encode this instruction. The sub-tree rooted at node `05` encodes all the valid operands of the instruction. Without any insight on the format of the instruction, one has to traverse in depth-first ordering the entire sub-tree and to assume that each path represents a different instruction. Then, for each traversed path, a test case must be generated. Our algorithm, by traversing only few paths of the sub-tree rooted at node `05`, is able to infer the format of the instruction: (i) the existence of the operand, (ii) which bytes of the instruction encode the opcode and which ones encode the operand, and (iii) the type of the operand. Once the instruction has been decoded (in the case of the example the opcode is `66 05` and it is followed by a 16-bit immediate), without having to traverse the remaining paths, our algorithm generates a minimal set of test cases with a very high coverage of all the possible behaviors of the instruction. These test cases are generated by fixing the bytes of the opcode and varying the bytes of the operand. The intent is to select operand values that more likely generate the larger class of behaviors (e.g., to cause an overflow or to cause an operation with carry). For example, for the opcode `66 05`, our algorithm decodes the instruction by exploring only 0.5% of the total number of paths and generates only 56 test cases. The optimized tree traversal is shown in Figure 3.3(b), where paths in gray are those that do not need to be explored. The heuristics on which our rudimentary, but

faithful, instructions decoder is built on is described in section 3.3.2. It is worth noting that, unlike traditional disassemblers, we decode instructions without any prior knowledge of their format. Thus, we can infer which bytes of an instruction represent the opcode, but we do not know which high-level instruction (e.g., `add`) is associated with the opcode.

3.3.2 The Decoder

The optimised traversal algorithm, just described in Section 3.3.1.2, requires the ability to decode an instruction, and to identify its opcode and operands. Such a task is undertaken by a specific module (less than 500 lines of code) which we named the decoder. The decoder uses the CPU as an oracle: given a sequence of bytes, the CPU tells us if that sequence encodes a valid instruction or not [43]. The decoding is trial-based: we mutate an executable sequence of bytes, we query the oracle to see which mutations are valid and which are not, and from the result of the queries we infer the format of the instruction. Mutations are generated following specific schemes that reflect the ones used by the CPU to encode operands [15].

In the following we briefly describe how the decoder infers the length of an instruction and the format of non-implicit operands, assuming to know only the encoding schemes used to encode operands.

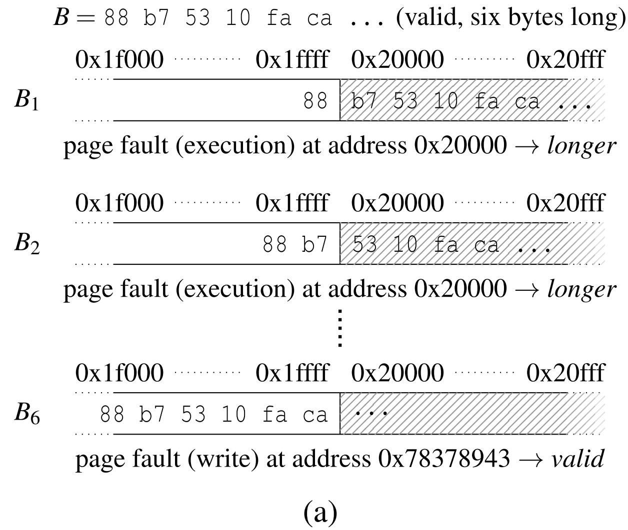
3.3.2.1 Determining Instruction Length

For determining the length of a given instruction the decoder exploits the fact that the CPU fetches, and decodes, the bytes of the instruction incrementally. Given an arbitrary sequence of bytes $B = b_1 \dots b_n$, the first goal is to detect if the bytes represent a valid instruction. The decoder executes the input string B in a specially crafted execution environment, such that every fetch of the bytes composing the instruction can be observed.

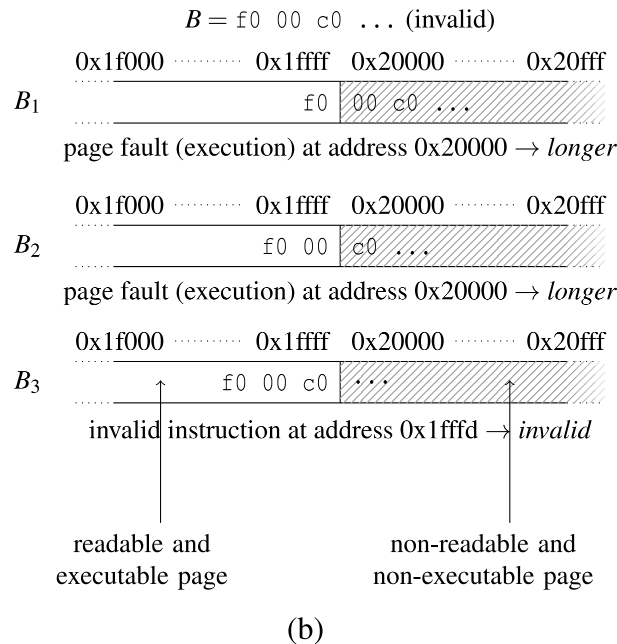
The decoder partitions B into subsequences of incremental length ($B_1 = b_1$, $B_2 = b_1 b_2$, $\dots$, $B_n = b_1 \dots b_n$) and then executes one subsequence after another, using single-stepping. The goal is to intercept the fetch of the various bytes of the instruction, which is achieved by placing the ith subsequence B_i (with $i = 1 \dots n$) in memory such that it overlaps two adjacent memory pages, m and m'. The first i bytes are located at the end of m, and the remaining $(n-i)$ bytes at the beginning of m'. The two pages have special permissions: m allows read and execute accesses, while m' prohibits any access. When the instruction is executed, the i bytes in the first page are fetched incrementally by the CPU. If the instruction is *longer* than i bytes, the CPU will try to fetch the next byte, $(i+1)^{th}$, and will raise a page fault exception (where the faulty address corresponds to the base address of

m') because the page containing the byte being read, m', is not accessible. In this case the decoder repeats the process with the string B_{i+1}, that is placing the $i+1^{th}$ bytes at the end of m and the remaining at m'. On the other hand, if the instruction contained in the page m has the correct length, it will be executed by the CPU without accessing the bytes in m'. In such a situation the instruction can be both valid and invalid. The instruction is *valid* if it is executed without causing any exception; it is also valid if the CPU raises a page fault (in this case the faulty address does not correspond to the base address of m') or a general protection fault exception. A page fault exception occurs if the instruction tries to read or write data from the memory; a general protection fault exception is raised if the instruction has improper operands. The instruction is *invalid* instead, if the CPU raises an illegal instruction exception. In both cases the decoder returns.

Figure 3.4 shows our CPU-assisted decoder in action on two different sequences of bytes, one valid and one invalid. The first sequence is $B =$ `88 b7 53 10 fa ca ...`, corresponding to the instruction `mov %dh, $0xcafa1053(%edi)`. The decoder allocates two adjacent memory pages and removes any permission from the second one. Then, it starts with the first subsequence $B_1 =$ `88`. The byte is positioned at the end of the page and then executed through single stepping. The CPU fetches and tries to decode the instruction but, since the instruction is longer than one byte, it tries to fetch the next bytes from the protected page, raising a page fault. The decoder detects the fault and concludes that the instruction is longer than one byte (in our example the faulty address is 0x20000, the base address of the second page). It repeats the procedure with $B_2 =$ `88 b7` and gets the same result. It tries again with B_3, B_4, B_5, and finally tries with six bytes. Since the instruction is six bytes long, the CPU executes the instruction without accessing the protected memory page. However, the instruction writes into the memory and thus causes a page fault. As in this case the faulty address (0x78378943) differs from the address of the protected page, our decoder can decide that the instruction is valid and that it is six bytes long. It is worth noting that a sequence of bytes cannot encode, at the same time, a valid instruction and a prefix of a longer instruction. Indeed, such a situation would be ambiguous for the CPU. The third byte sequence in the example of Figure 3.4(b) is $B =$ `f0 00 c0 ...` and represents an invalid instruction. Exactly as before, our decoder executes the first two subsequences B_1 and B_2 and detects that the instruction is potentially longer because the CPU fetches a third byte from the protected page. When B_3 is executed, the CPU does not fetch more bytes but instead raises an illegal instruction exception, testifying that B_3 is neither a valid instruction, nor a valid prefix for longer instructions.



(a)



(b)

Figure 3.4: Computation of the length of instructions using our CPU-assisted instruction decoder: (a) valid and (b) invalid instructions.

3.3.2.2 Decoding Non-implicit Operands

Once the decoder finds the length of an instruction the decoder tries to infer the type and the value of the non-implicit operands of the instruction (i.e., the operands that are not implicitly encoded in the opcode of the instruction). The

technique used by our decoder to achieve this goal is an extension of the technique described in the previous paragraphs. Currently, our CPU-assisted decoder is capable of decoding addressing-form specifier operands and immediate operands.

Any Intel x86 instruction (Figure 2.1) is composed of an optional prefix, an opcode, and optional operands. To ease the presentation we assume that the instructions have no prefix; in practice, prefixes are detected using a white-list and considered part of the opcode. Given an instruction, encoded by the sequence of bytes $B = b_1 \dots b_n$, the format of the operands is detected by performing a series of tests on some instructions derived by changing the bytes of B that follow the opcode and represent the operands of the instruction. If the opcode is j bytes long, the remaining $n - j$ bytes represent the operands. Each type of operand is encoded using a different encoding: immediate operands (*Imm*) are encoded as is, addressing-form specifier operands (*Addr*) are encoded using `ModR/M` and `SIB` encoding, and $Imm \cup Addr \neq Imm \cap Addr$ (i.e., an immediate operand does not necessarily represent a valid addressing-form specifier operand, and vice versa). Therefore, given an instruction encoded by the sequence of bytes $B = b_1 \dots b_n$, we expect a new sequence $B' = b_1 \dots b_j b'_{j+1} \dots b'_m$, where $b'_{j+1} \dots b'_m$ represents a new operand of the same type of $b_{j+1} \dots b_m$, to be valid. Contrarily, we expect another sequence of bytes $\overline{B} = b_1 \dots b_j \overline{b_{j+1}} \dots \overline{b_m}$, where $\overline{b_{j+1}} \dots \overline{b_m}$ represent an operand of a different type, to be invalid. Therefore, if an instruction with a j bytes long opcode has an immediate operand, then the following holds:

$$\forall b'_{j+1} \dots b'_m \in Imm, B' = b_1 \dots b_j b'_{j+1} \dots b'_m \text{ is valid.}$$

In other words, the bytes following the opcode encode an immediate operand if the combination of the opcode with all the possible immediate operands always gives valid instructions. Fortunately, with few tests it is possible to estimate if the previous equation holds. In fact, it is sufficient to verify if it holds for a small number of operands in $Imm \setminus Addr$. The same applies for an instruction with an addressing-form specifier operand. Our current prototype of the decoder uses only five tests to decode addressing-form specifier operands and four to detect 32-bit immediate operands. Basically, in order to infer if an instruction refers to an operand in memory, we use specific configurations of the `ModR/M` and `SIB` fields (e.g., `[EAX], [EAX]+disp, [EBP]+disp`, etc.). Since the opcode can have a variable length (from one to three bytes), our CPU-assisted decoder performs the aforementioned tests with opcodes of incremental length (i.e., $j = 1, 2, 3$).

Figure 3.5 shows some of the tests performed by our CPU-assisted instruction decoder to infer the format of the operands of two instructions: the first instruction has an addressing-form specifier operand and the second one a 32-bit immediate operand. For the first instruction, the decoder initially assumes that the opcode is one byte long, and performs the analysis of the remaining bytes to detect if they

$B =$ `88 b7 53 10 fa ca`
`mov %dh, $0xcafa1053(%edi)`

0x1f000 ⋯ 0x1ffff 0x20000 ⋯ 0x20fff
B'_2 `88 00` | `53 10 fa ca`
page fault (write) at address 0x00 → *valid*

0x1f000 ⋯ 0x1ffff 0x20000 ⋯ 0x20fff
B'_3 `88 40 00` | `10 fa ca`
page fault (write) at address 0x000 → *valid*

0x1f000 ⋯ 0x1ffff 0x20000 ⋯ 0x20fff
B'_4 `88 44 25 00` | `fa ca`
page fault (write) at address 0x00 → *valid*

⋮

0x1f000 ⋯ 0x1ffff 0x20000 ⋯ 0x20fff
B'_7 `88 04 25 00 00 00 00` |
page fault (write) at address 0x00 → *valid*
test passed → operand is an addressing-form specifier

(a)

$B =$ `05 12 34 56 78`
`add $0x78563412, %eax`

0x1f000 ⋯ 0x1ffff 0x20000 ⋯ 0x20fff
B'_2 `05 00` | `34 56 78`
page fault (execution) at address 0x20000 → *longer*
test failed → operand is not an addressing-form specifier

0x1f000 ⋯ 0x1ffff 0x20000 ⋯ 0x20fff
B'_5 `05 00 00 00 01` |
no exception → *valid*

0x1f000 ⋯ 0x1ffff 0x20000 ⋯ 0x20fff
B''_5 `05 00 00 00 02` |
no exception → *valid*

⋮

0x1f000 ⋯ 0x1ffff 0x20000 ⋯ 0x20fff
$B'''^{\cdots}_5$ `05 00 00 00 255` |
no exception → *valid*
test passed → operand is a 32-bit immediate

(b)

Figure 3.5: Decoding of non-implicit operands using our CPU-assisted instruction decoder: instructions with (a) addressing-form specifier operand and (b) immediate operand.

encode an addressing-form specifier operand. To do that it combines the opcode `88` with other valid addressing-form specifier operands of variable length, some of which cannot be interpreted as immediate operands. The first test consists of replacing the alleged operand with a single byte operand and in executing the resulting string. The CPU successfully executes the instruction. The same procedure is repeated with operands of different length (two, three, and seven bytes). All the sequences of bytes are found to encode valid instructions; every execution of the tested instructions raise a page fault exception where the faulty address does not correspond to the base address of the protected page. Therefore, the input instruction is composed of a single byte opcode followed by an addressing-form specifier operand (`b7 53 10 fa ca`, in Figure 3.5). The same procedure is applied also to the second instruction. The addressing-form specifier operand decoding fails, so the decoder attempts to verify whether the last four bytes of the instruction encode a 32-bit immediate. All tests performed are passed.

3.3.3 Test Case Execution

Given a test case, we have to execute it both on the physical and emulated CPUs and then compare their state at the end of the execution. In order to perform such a task we have developed two different applications, the first one denoted by E runs on the emulator and the second one, denoted by P will run on the physical CPU as a user space application. Initially, we start the execution of the test case on the emulator. As soon as the initialization of the state of the emulator is completed, it is replicated to the physical CPU. As registers and memory are initialized with random values, replication is required to guarantee that test cases are executed on the physical and emulated environments starting from the same initial state. Then, the code of the test case is executed in the two environments and, at the end of the execution, we compare the final state. In the remainder of this section we describe the main steps performed for the execution of a test case and we will also provide details on the strategy we adopted for instrumenting the emulator and the physical environment in order to execute respectively the programs E and P. For simplicity, the details that follow are specific for the testing of process emulators. Nonetheless, the implementation for testing whole-system emulators only requires the addition of introspection capabilities to isolate the execution of the test case program [53].

3.3.3.1 Executing a Test Case

The execution flow of a test case is summarized in Figure 3.6 and described in detail in the following paragraphs, where the following notation will be adopted. The state of the emulator $\mathscr{C}_E$ prior and after the execution of a test case respec-

tively $s_E = (pc_E, R_E, M_E, E_E)$ and $s'_E = (pc'_E, R'_E, M'_E, E'_E)$. Similarly, for $\mathscr{C}_P$, we use respectively $s_P = (pc_P, R_P, M_P, E_P)$ and $s'_P = (pc'_P, R'_P, M'_P, E'_P)$.

Setup of the Emulated Execution Environment The CPU emulator is started and it begins to execute the program E generating and executing the test case ($L_{\mathrm{E}1}$) until the state of the environment is completely initialized ($L_{\mathrm{E}2}$). In other words, E is executed without interference until the execution reaches pc_E, i.e., the address of the code of the test case (see line 19, Figure 3.2). E initializes the emulator memory by mapping a file filled with random data. For simplicity, the same file is mapped multiple times at consecutive addresses until the entire user-portion of the address space is allocated. To avoid a useless waste of memory, the file is lazily mapped in memory, such that physical memory pages are allocated only if they are accessed. As we discussed in section 3.3.1.1, CPU registers are also initialized with random values.

Setup of the Physical Execution Environment When the state of the emulated environment has been set up (i.e., when the execution has reached pc_E), the initial state, $s_E = (pc_E, R_E, M_E, E_E)$, can be replicated into the physical environment. The emulator notifies and transfers the state of the CPU registers to P ($L_{\mathrm{E}3}$). Initially, the exception state E_E is always assumed to be $\perp$. Note that the memory state of the physical CPU M_P is not synchronized with the emulated CPU. At the beginning, only the memory page containing the code of the test case is copied into the physical environment ($L_{\mathrm{P}1}$ and $L_{\mathrm{E}4}$). The remaining memory pages are instead synchronized on-demand the first time they are accessed, as it will be explained in detail in the next paragraph. At this point we have that $R_E = R_P$, $E_E = E_P = \perp$, but $M_E \neq M_P$ (the only page that is synchronized is the one with the code).

Test Case Execution on the Physical CPU The execution of the code of the test case on the physical CPU starts, beginning from program address $pc_P = pc_E$ ($L_{\mathrm{P}3}$). P besides an initialization routine, to set up the execution environment, also contains a finalization routine, to save the content of the registers; moreover, test cases instructions are patched to avoid unwanted control transfers. For further details see section 3.3.3.3. During the execution of the code, the following situations may occur:

i execution of the code of the test case terminates;

ii a page-fault exception caused by an access to a missing page occurs;

iii a page-fault exception caused by a write access to a non-writable page occurs;

iv any other exception occurs.

Situation (i) indicates that the entire code of the test case is executed successfully. That means that the instruction in the test case was valid and did not generate any fatal CPU exception. The first type of page-fault exceptions (ii) allows us to synchronize lazily the memory containing the data of the test case at the first access. During the initialization phase (L_{P2}) all the memory pages of the physical environment, but that containing the code (and few others containing the code to run the logic), are protected to prevent any access. Consequently, if an instruction of the test case tries to access the memory, we intercept the access through the page fault exception and we retrieve the entire memory page from the emulated environment (L_{P4} and L_{E5}). All data pages retrieved are initially marked as read-only to catch future write accesses. After that, the execution of the code of the test case on the physical CPU is resumed (L_{P5}). The second type of page-fault exceptions (iii) allows us to intercept write accesses to the memory. Written pages are the only pages that can differ from one environment to the other. Therefore, after a faulty write operation we flag the memory page as written. Then, the page is marked as writable and the execution is resumed (L_{P6} and L_{P7}). Obviously, depending on the code of the test case, situations (ii) and (iii) may occur repeatedly or may not occur at all during the analysis. Finally, the occurrence of any other exception (iv) indicates that the execution of the code of the test case cannot be completed because the CPU is unable to execute an instruction. When the execution of the code of the test case on the physical CPU terminates, because of (i) or (iv), P regains the control of the execution, immediately saves the state of the environment for future comparisons (L_{P8}), and restores the state of the CPU prior to the execution of the test case.

Test Case Execution on the Emulated CPU The execution of the code of the test case in the emulated environment, previously stopped at pc_E (L_{E2}), can now be safely resumed. The execution of the code in the emulated environment must follow the execution in the physical environment and cannot be concurrent with it. This is because in the physical environment the state of the memory is synchronized on-demand and thus the initial state of the memory M_E must remain untouched until the physical CPU completes the execution of the test case. When this happens the execution is resumed and it terminates when all the code of the test case is executed or an exception occurs (L_{E6}).

Comparison of the Final State When the emulator and the physical environments have completed the execution of the test case we can compare their state ($s'_E = (pc'_E, R'_E, M'_E, E'_E)$ and $s'_P = (pc'_P, R'_P, M'_P, E'_P)$). The comparison is performed by P. The emulator notifies P and then transfers the program counter

pc'_E, the current state of the CPU registers R'_E, and the exception state E'_P (L_{E7}). To compare s'_E and s'_P it is not necessary to compare the entire address space: P fetches only the contents of the pages that have been marked as written (L_{P10} and L_{E8}). At this point s'_E is compared with s'_P (L_{P11}). If s'_E differs from s'_P, we record the test case and the difference(s) produced.

3.3.3.2 Embedding the Logic in the CPU Emulator

Program E is run directly in the emulator under analysis. The emulator is extended to include the code of E. We embed the code leveraging the instrumentation API provided by the majority of the emulators. The main functionalities of the embedded code are the following. First, it allows to intercept the beginning and the end of the execution of each instruction (or basic block, depending on the emulator) of the emulated program. If the code of the test case contains multiple instructions, all basic blocks (or instructions) are intercepted and contribute to the testing. We assume the code used to initialize the environment is always correctly emulated and thus we do not test it nor we intercept its execution. Second, the embedded code allows to intercept the exceptions that may occur during the execution of the test case. Third, it provides an interface to access the values of the registers of the CPU and the contents of the memory of the emulator.

3.3.3.3 Running the Logic on the Physical CPU

On the physical CPU, the test case is run through a user-space program that implements the various steps described in 3.3.3.1. An initialization routine (L_{P2} in Figure 3.6), is used to set up the registers of the CPU, to register signal handlers to catch page faults and the other run-time exceptions that can arise during the execution of the test case, and to transfer the control to the code of the test case. The code of the test case is executed as a shellcode [54] and consequently we must be sure it does not contain any dangerous control transfer instruction that would prevent us from regain the control of the execution (e.g., jumps, function calls, system calls). Given the approaches we use to generate the code of the test cases, we cannot prevent the generation of such dangerous test cases. Therefore, we rely on a traditional disassembler to analyze the code of the test case, identify dangerous control transfer instructions, and patch the code to regain the control of the execution (e.g., by modifying the target address of direct jump instructions)[7]. To prevent endless loops caused by failures of this analysis, we put a limit on the maximum CPU time available for the execution of a test case and we interrupt the execution if the limit is exceeded. In the current implementation, this limit is set

[7] If the disassembler failed to detect dangerous control transfer instructions, we could not be able to regain the control of the execution properly.

Table 3.2: Results of the evaluation: number of distinct mnemonic opcodes (OP) and number of test cases (TC) that triggered deviations in the behavior between the tested emulators and the baseline physical CPU.

Deviation type		QEMU		Valgrind		Pin		BOCHS		JPC	
		OP	*TC*	*OP*	*TC*	*OP*	*TC*	*OP*	*TC*	*OP*	*TC*
R	*CPU flags*	39	1362	13	684	22	2180	2	2686	33	4088
	CPU general	3	142	8	141	3	18	8	8	27	657
	FPU	179	41738	157	39473	0	0	71	1631	185	43024
M	*memory state*	34	1586	10	420	0	0	1	2	46	2122
E	*not supported*	2	1120	334	11513	2	12	0	0	8	1998
	over supported	97	1859	10	716	0	0	5	8	124	1930
	other	126	6069	41	6184	20	34	45	113	132	5935
	Total	**405**	**53926**	**529**	**59135**	**43**	**2245**	**130**	**4469**	**482**	**59354**

to 5s, and has been determined experimentally to guarantee detection of endless loops. At the end of the code of the test case we append a finalization routine (L_{P8} in Figure 3.6), that is used to save the contents of the registers for future comparison, to restore their original contents, and to resume the normal execution of the remaining steps of the logic. Exceptions other than page-faults interrupt the execution of the test case. The handlers of these exceptions record the exception occurred and overwrite the faulty instruction and the following ones with nops, to allow the execution to reach the finalization routine to save the final state of the environment.

In the approach just described the program P and the test case share the same address space. Therefore, the state of the memory in the physical environment differs slightly from the state of the memory in the emulated environment: some memory pages are used to store the code and the data of the user-space program, through which we run the test case. If the code of the test case accesses any of these pages, we would notice a spurious difference in the state of the two environments. Considering that the occurrence of such event is highly improbable, we decided to neglect this problem, to avoid complicating the implementation.

3.4 Evaluation

This section presents the results we obtained by testing five IA-32 emulators with EmuFuzzer: three process emulators (QEMU, Valgrind, and Pin) and two system emulator (BOCHS and JPC). Specifically, we chose QEMU and BOCHS because they are the most widely used IA-32 emulators, Valgrind and Pin because, despite them being dynamic instrumentation tools, their internal architecture resembles

the architecture of a traditional emulator, and finally JPC because it is an IA-32 emulator fully developed in Java language and therefore portable on several platforms and devices (e.g., mobile devices).

We generated a large number of test cases, evaluated their quality, and fed them to the five emulators. None of the emulators tested turned out to be faithful. In each of them we found different classes of defects: small deviations in the contents of the status register after arithmetical and logical operations, improper exception raising, incorrect decoding of instructions, and even crash of the emulator. Our experimental results lead to the following conclusions: (i) developing a CPU emulator is actually very challenging, (ii) developers of these software would highly benefit from specialized testing methodology, and (iii) EmuFuzzer proved to be a very effective tool for testing CPU emulators.

3.4.1 A Glimpse at the Implementation

The current EmuFuzzer implementation consists of three interconnected components: a *coordinator* and two *drivers* (one for the physical CPU, and one for the emulator under analysis). The coordinator supervises the execution of a test case. The driver that controls the execution on the physical CPU is independent from any specific emulator. On the contrary, the driver for CPU emulator augments a specific emulator with the features needed to intercept the execution of a single instruction and to inspect the execution state; this driver is obviously emulator-specific, and we implemented a different emulator driver for each CPU emulator we considered in our experiments.

For a given test case, s, the coordinator first leverages the emulator driver to set up the emulated execution environment: CPU registers and memory locations are initialized as specified by s. Subsequently, as just mentioned in section 3.3.3, the coordinator starts executing the test case on the physical processor. Such an execution may require the setting of some CPU registers or memory locations, that are thus fetched from the emulated environment and replicated into the physical one. Once the execution of the test case completes, the processor final state $\delta_{\mathscr{C}_P}(s)$ is dumped to a file. At this point the coordinator starts the execution of s on the emulator. Also in this case the final state of the computation, $\delta_{\mathscr{C}_E}(s)$ will be dumped to a file. Then, the coordinator compares $\delta_{\mathscr{C}_P}(s)$ and $\delta_{\mathscr{C}_E}(s)$.

The coordinator is written in Python ($\sim$ 750 non-comment lines of code), while the CPU driver consists of roughly 1200 non-comment lines of C++ code. Finally, emulator drivers are written in the same language of the target emulator, typically C. On average, an emulator driver requires about 450 non-comment lines of code (of these, 350 lines are emulator-independent and are shared between all emulator drivers). Communication between the coordinator and the drivers relies on the XML-RPC protocol.

3.4.2 Experimental Setup

We performed the evaluation of our testing methodology and tool using an Intel Pentium 4 (3.0 GHz), running Debian GNU/Linux with kernel 2.6.26, as baseline physical CPU. The physical CPU supported the following features: MMX, SSE, SSE2, and SSE3. We tested the following release of each emulator, namely: QEMU 0.9.1, Valgrind 3.3.1, Pin 2.5-23100, JPC 2.4, and BOCHS 2.3.7. The features of the physical machine were compatible with the features of the tested emulators with few exceptions, which we identified at the end of the testing, using a traditional disassembler, and ignored (for example, BOCHS also supports SSE4).

3.4.3 Evaluation of Test Case Generation

We generated about 3 million test cases, 70% of which using our CPU-assisted algorithm and the remaining 30% randomly. We empirically estimated the completeness of the set of instructions covered by the generated test cases by disassembling the code of the test cases, by counting the number of different instructions found (operands were ignored), and by comparing this number with the total number of mnemonic instructions recognized by the disassembler. The randomly generated test cases covered about 75% of the total number of instructions, while the test cases generated using our CPU-assisted algorithm covered about 62%. Overall, about 81% of the instructions supported by the disassembler were included in the test cases used for the evaluation. It is worth noting that in several cases our test cases contained valid instructions not recognized by the disassembler.

The implementation of our CPU-assisted algorithm is not complete and lacks support for all instructions with prefixes. For example, currently our algorithm does not generate test cases involving instructions operating on 16-bits operands. We have empirically estimated that instructions with prefixes represent more than 25% of the instructions space. Therefore, a complete implementation of the algorithm would allow to achieve a nearly total coverage. We speculate that the high coverage of randomly generated test cases is due to the fact that the IA-32 instruction set is very dense and consequently a random bytes stream can be interpreted as a series of valid instructions with high probability. Nevertheless, during our empirical evaluation we reached a local optimum from which it was impossible to move away, even after having generated hundreds of thousands of new test cases. The CPU-assisted algorithm instead does not suffer this kind of problem: a complete implementation would allow to generate a finite number of test cases exercising all instructions in multiple corner cases.

3.4.4 Testing of IA-32 Emulators

The five CPU emulators were tested using a small subset (∼10%) of the generated test cases, selected randomly. The whole testing took about a day for all the emulators but JPC, at the speed of around 15 test cases per second (JPC alone took several days to run all the test cases). Table 3.2 reports the results of our experiments. Behavioral differences found are grouped into three categories: CPU registers state (R), memory state (M), and exception state (E). Differences in the state of the registers are further separated according to the type of the registers: status (*CPU flags*), general purpose and segment (*CPU general*), and floating-point (*FPU*). Differences in the exception state are separated in: legal instructions not supported by the emulator (*not supported*), illegal instructions valid for the emulator (*over supported*), and other deviations in the exception state (*other*). As an example, the last class includes instructions that expect aligned operands but execute without any exception even if the constraint is not satisfied. For each emulator and type of deviation, the table reports the number of distinct mnemonic opcodes leading to the identification of that particular type of deviation (*opcodes*) and the number of test cases proving the deviation (*test cases*). It is worth pointing out that different combinations of prefixes and opcodes are considered as different mnemonic opcodes. For each distinct opcode that produced a particular type of deviation, we verified and confirmed manually the correctness of at least one of the results found.

The results demonstrate the effectiveness of the proposed testing methodology. For each emulator we found several mnemonic opcodes not faithfully emulated: 405 in QEMU, 529 in Valgrind, 43 in Pin, 130 in BOCHS and 482 in JPC. It is worth noting that some of the deviations found might be caused by too lax specifications of the physical CPU. For example, the manufacturer documentation of the `add` instruction precisely states the effect of the instruction on the status register, while the documentation of the `and` instruction only states the effect on some bits of the status register, while leaving undefined the value the remaining bits [15]. Our reference of the specification is the CPU itself and consequently, with respect to our definition of faithful emulation, any deviation has to be considered a tangible defect. Indeed, *for each deviation discovered by EmuFuzzer it is possible to write a program that executes correctly in the physical CPU, but crashes in the emulated CPU (or vice versa).* We manually transformed some of the problematic test cases into this kind of programs and verified the correctness of our claim. The remarkable number of defects found also witnesses the difficulty of developing a fully featured and specification-compliant CPU emulator and motivates our conviction about the need of a proper testing methodology.

The following paragraphs summarize the defects we found in each emulator. The description is very brief because the intent is not criticize the implementation

of the tested emulators, but just to show the strength of EmuFuzzer at detecting various classes of defects.

In [55] we release all the improper behaviors we detected in the emulators supported by EmuFuzzer. Developers were informed about the defects found in their emulators, providing them with the corresponding test cases.

3.4.4.1 QEMU

A number of arithmetical and logical instructions are not properly executed by the emulator because of an error in the routine responsible for decoding certain encoding of memory operands (e.g., `or %edi, 0x67(%ebx)` encoded as `08 7c e3 67`); the instructions reference the wrong memory locations and thus compute the wrong results. The emulator accepts several illegal combinations of prefixes and opcodes and executes the instruction ignoring the prefixes (e.g., `lock fcos`). Floating-point instructions that require properly aligned memory operands are executed without raising any exception even when the operands are not aligned, because the decoding routine does not perform alignment checking (e.g., `fxsave 0x00012345`). Segments registers, which are 16 bits wide, are emulated as 32-bit registers (the unused bits are set to zero), thus producing deviations when they are stored in other 32-bits registers and in memory (e.g., `push %fs`). Some arithmetic and logical instructions do not faithfully update the status register. Finally, we found sequences of bytes that freeze and others that crash the emulator (e.g., `xgetbv`).

3.4.4.2 Valgrind

Some instructions have multiple equivalent encodings (i.e., two different opcodes encode the same instruction) but the emulator does not recognize all the encodings and thus the instructions are considered illegal (e.g., `addb $0x47, %ah` with opcode `82`). Several legal privileged instructions, when invoked with insufficient privileges, do not raise the appropriate exceptions (e.g., `mov (%ecx), %cr3` raises an illegal operation exception instead of a general protection fault). On the physical CPU, each instruction is executed atomically and, consequently, when an exception occurs the state of the memory and of the registers correspond to the state preceding the execution of the instruction. On Valgrind instead, instructions are not executed atomically because they are translated into several intermediate instructions. Consequently, if an exception occurs in the middle of the execution of an instruction, the state of the memory and of the registers might differ from the state prior to the execution of the instruction (e.g., `idiv (%ecx)` when the divisor is zero). As in QEMU, some logical instructions do not faithfully update the status register.

3.4.4.3 Pin

Not all exceptions are properly handled (i.e., trap and illegal instruction exceptions); Pin does not notify the emulated program about these exceptions. Several legal instructions that raise a general protection fault on the physical CPU are executed without generating any exception on Pin (e.g., `add %ah, %fs:(%ebx)`). When segment registers are stored (and removed) in the stack, the stack pointer is not updated properly: a double-word should be reserved on the stack for these registers, but Pin reserves a single word (e.g., `push %fs`). The FPU appears to be virtualized (i.e., the floating-point code is executed directly on the physical FPU) and, as expected, no deviation is detected in the execution of FPU instructions. As in Valgrind and QEMU, some logical instructions do not faithfully update the status register.

3.4.4.4 BOCHS

Certain floating-point instructions alter the state of some registers of the FPU and other instructions compute results that differ from those computed by the FPU of the physical CPU (e.g., `fadd %st0, %st7`). If an exception occurs in the middle of the execution of an instruction manipulating the stack, the initial contents of the stack pointer corresponds to that we would have if the instruction were successfully executed (e.g., `pop 0xffffffff`). Some instructions do not raise the proper exception (e.g., `int1` raises a general protection fault instead of a trap exception). As in Valgrind, QEMU, and Pin, some logical instruction do not faithfully update the status register, although the number of such instruction is smaller than the number of instructions affected by this problem in the other emulators.

3.4.4.5 JPC

Conversely to the other emulators, which are written in C and C++, JPC is fully developed in Java. It turned out that one of the main problems we had to deal with for testing this emulator was its poor performances executing test cases: JPC is approximately 75% slower than any other emulator. A description of the main deviations found follows. Segment registers, which are 16-bits wide, are emulated as 32-bit registers. This implies deviations when segment registers are stored in other 32-bits registers and in memory (e.g., `push %gs`). Several legal privileged instructions, when invoked with insufficient privileges, do not raise the appropriate exceptions (e.g., `mov (%ecx), %cr0` and `mov %cr3, %eax` are correctly executed when the CPU is in user mode without raising a general protection fault exception). As in QEMU, the emulator accepts illegal combination of prefixes and executes the instruction ignoring them (e.g., `lock fcos`). Moreover, not all exceptions are properly handled (i.e., illegal instruction exceptions) and some

instructions do not raise the appropriate exception. To conclude, we found several sequences of bytes that crash the emulator (e.g., `bound %eax, (%ebx)` and `int1`).

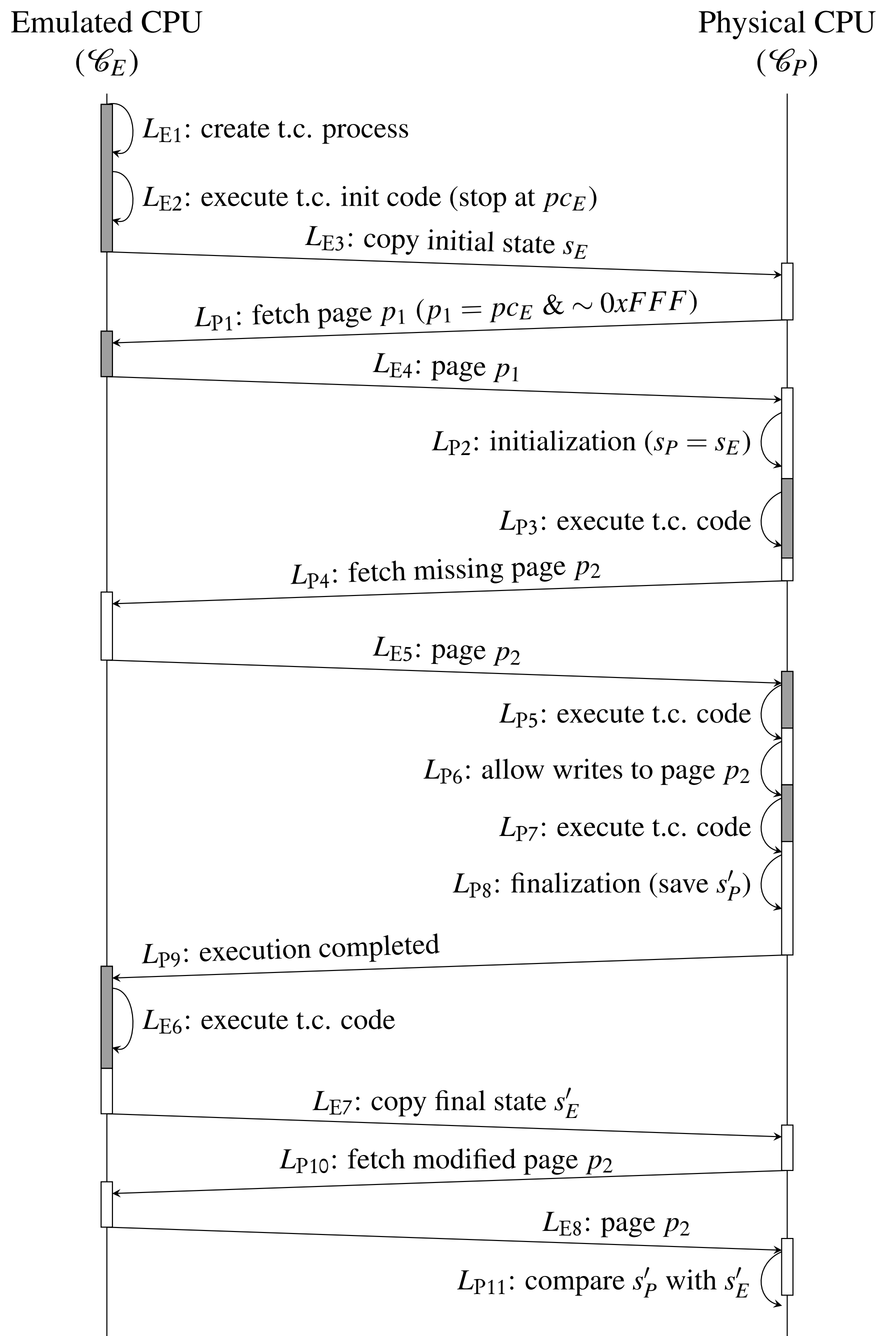


Figure 3.6: Logic of the execution of a test case (t.c., for short). ■ denotes the execution of the test case and □ denotes the execution of the code of the logic.

4

On Reconstructing Android Malware Behaviors

With more than 500 million of activations reported in Q3 2012, Android mobile devices are becoming ubiquitous and trends show that such a pace is unlikely slowing down [56]. Android devices are extremely appealing: powerful, with a functional and easy-to-use user interface to access sensitive user and enterprise data, they can easily replace traditional computing devices, especially when information is mostly consumed rather than produced.

Application marketplaces, such as Google Play and the Apple App Store, drive the entire economy of mobile applications. For instance, with more than 600,000 applications installed, Google Play has generated revenues of about 237M USD per year [57]. Such a wealth and quite unique ecosystem with high turnovers and access to sensitive data have unfortunately also attracted the interests of cyber-criminals, with malware now hitting Android devices at an alarmingly rising pace. Users privacy breaches (e.g., access to address book and GPS coordinates) [58], monetization through premium SMS and calls [58], and colluding malware to bypass 2-factor authentication schemes [59] are all real threats rather than a fictional forecasting. Recent studies back easily such statements up, reporting how mobile marketplaces have been abused to host malware or legitimate-resembling applications embedding malicious components [60].

Unfortunately, the nature of Android applications makes it hard, if not impossible, to rely on existing VM-based dynamic malware analysis systems as is. In fact, Android applications are generally written in the Java programming language and executed on top of the Dalvik virtual machine [61], but native code invocation is however possible via JNI or Linux ELF binary execution. This mixed environment seems to suggest the need to reconstruct and keep in sync out-of-the-box semantics through virtual machine introspection (VMI) [62] for both the OS and Dalvik views, as recently shown in [63]. On the one hand, OS-level semantics (e.g., writing to a file, executing a program) would allow to characterize JNI or

native ELF-induced behaviors, while Dalvik-level semantics would enable to disclose high-level Android-specific behaviors (e.g., sending an SMS). While true in principle, we observe that even high-level Android-specific behaviors are indeed achieved via system call invocations, underneath. In fact, as described later, Android applications may interact *with the system* via well-defined system call-initiated IPC and RPC invocations to carry out their tasks.

In this work we present CopperDroid, an approach built on top of QEMU [64] to *automatically* perform out-of-the-box dynamic behavioral analysis of Android malware. To this end, CopperDroid presents a *unified* analysis to characterize low-level OS-specific (e.g., opening and writing to a file, executing a program) and high-level Android-specific (e.g., accessing personal information, sending an SMS) behaviors. In particular, based on the observation that such behaviors are all achieved through the invocation of system calls, CopperDroid's VMI-based system call-centric analysis faithfully describes Android malware behavior whether it is initiated from Java, JNI or ELF code.

A preliminary description of CopperDroid, focused on introducing our system call-centric analysis as well as its effectiveness evaluation on well-known yet outdated Android malware datasets appeared recently in our workshop paper [65]. Conversely, in this dissertation, we present our mature research effort and improvements over [65] whose contributions can be summarized as follows:

1. We describe the design and implementation of a *unified* dynamic analysis technique to characterize the behavior of Android malware. Our analysis is able to automatically describe low-level OS-specific and high-level Android-specific behaviors of Android malware by observing and analyzing system call invocations, including IPC and RPC interactions—of paramount importance on Android—carried out as system calls underneath. To automate the analysis of Android's IPC and RPC channel, we design and implement a novel technique that avoids—or reduces to a bare minimum in a limited number of cases—the amount of manual analysis needed to reconstruct remote invocations and their parameters.

2. Based on the observation that Android applications are inherently user-driven and feature a number of implicit but well-defined entry points, we describe the design and implementation of a stimulation approach aimed at disclosing additional malware behaviors.

3. We build a data dependency graph over the set of observed system calls, and perform forward slicing to select all instructions dependent on a particular `open`-related system call [66]. An interesting side-effect of such an analysis is the ability to automatically recreate the resource associated with a stream of sliced system calls, which, depending on the resource, can be fed back to

CopperDroid (e.g., Android app created at run-time), downloaded for further inspection, or submitted to other analysis systems.

4. We provide a thorough evaluation of CopperDroid's analysis on more than 1,200 malware samples belonging to 49 Android malware families as provided by the Android Malware Genome Project [67], about 400 samples over 12 Android malware families from the Contagio project [68], and more than 1,300 samples from McAfee, divided in roughly 115 families. Our experiments show that CopperDroid is able to *automatically* and *faithfully* describe the behavior of the samples in our datasets. Furthermore, CopperDroid confirms the importance of a proper malware stimulation approach (e.g., sending SMS, placing calls), which allowed us to disclose an average of, respectively: 28% of unique additional behaviors on 60% of malware samples in the first set, 22% on more than 70% of samples in the second set, and 28% on 61% of samples of the last set.

We believe CopperDroid's *unified* analysis contributes effectively to improve the state-of-the-art in analyzing the behavior of Android malware.

As further described in Section 4.3.6, CopperDroid relies on a simple-yet-effective stimulation technique that is able to improve basic dynamic analysis coverage and discover additional behaviors with low overheads.

Although a non-negligible implementation effort, we however consider the framework we developed and briefly describe in Section 4.3 as a mere yet necessary mechanism to carry out our actual contributions, i.e., CopperDroid's VMI-based system call-centric analysis—whose automatic IPC/RPC dissection is a key aspect—malware stimulation approach, and evaluation on large data sets.

Availability

CopperDroid is available at `http://copperdroid.isg.rhul.ac.uk`, where users can submit samples to be analyzed. Results contains behavioral analysis (both in HTML and JSON format, for easy parsing) and many ancillary information.

4.1 The Android System

Android applications are typically written in the Java programming language and then deployed as Android Packages archive (APKs). Every APK is considered to be a self-contained application that can be logically decomposed into one or more components. Each component is generally designed to fulfill a specific application task (e.g., GUI-related actions, notification receiver) and it is invoked either by the user or the OS.

According to the Android security model [69], each application runs in a separate userspace process, as an instance of the Dalvik virtual machine (DVM) [61], usually with a distinct user and group ID.

Applications must declare upfront the set of permissions (usually associated with additional group IDs) they wish to use. The user is informed at installation-time about the permissions the application is asking for, which gives him the final possibility to grant or deny the application installation. Because Android applications can also execute native code(e.g., via JNI), a number of trusted Android system services, along with the kernel, take care of starting a policy validation mechanism (which eventually will grant or deny specific permissions) [70]. Android applications are executed within their own instance of the DVM, without any possibility (except in the case of exploits) to get out of it and influence other applications. At a lower level, this corresponds to having a distinct process for each application. By doing so, the Android system creates a secure environment in which applications are sandboxed and strictly checked, against a permission system that will be later explained, and cannot access other applications and the system itself, without explicitly requiring it.

Permission-based security systems are generally effective, but understanding all the complexities of a fine-grained permission system may force developers to carelessly request too many permissions, exposing users to unneeded warnings and potentially loosening the effectiveness of the policy enforcement process [70, 71]. Furthermore, Barrera *et al.* reported that a number of privileges that can be specified in an APK are too coarse-grained while others are too fine-grained and hard to interpret [72]. This may lead to a misuse of the underlying privilege systems that may potentially lower the overall level of security of Android applications.

4.1.1 Application components

Although isolated within their own sandboxed environment, Android applications can interact with other applications, and with the system, through a well-defined API. A number of components can make up an application. In particular, Android defines activities, services, content providers, and broadcast receivers.

Activities provide a window the user can interact with. For instance, GUI elements needed to write a text, view a map, or send an email are all provided by activities. Services are similar to Unix dæmons; they run in the background and do not provide GUI element nor user interactions. For instance, a typical service component performs asynchronous network operations. Content providers define a storage-agnostic abstraction to transparently access data. They also perform access control, defining who can access data and how. Finally, broadcast receivers are components that listen (and respond) to broadcast events from the system.

```
<uses-permission android:name="[...].RECEIVE_SMS" />
<uses-permission android:name="[...].INTERNET" />
...
 <receiver android:name=".SMSReceiver">
  <intent-filter>
   <action android:name="..Telephony.SMS_RECEIVED" />
...
```

Figure 4.1: Example of Manifest file.

Whenever such events occur, the receiver is notified by the system and performs operations in response to the event (e.g., hide user notifications upon reception of SMS messages).

Activities, services, and broadcast receivers are activated by intents, i.e., asynchronous messages exchanged between individual components to request an action. Activity and service intents specify actions to be performed. Conversely, broadcast receiver intents define the received event and are delivered to the interested broadcast receivers.

4.1.2 Manifests

Android manifests are XML files that must be included in every APK. A manifest declares application components as well as the set of permissions the application requests along with the hardware and software features the application uses. In addition, a manifest may include intent filters, i.e., the set of intents the application is willing to handle.

Figure 4.1 reports a stripped-down Android manifest of a fictional but realistic application. The manifest shows the application requires permission to receive SMS and to access the Internet. Furthermore, it declares a broadcast receiver component (class `SMSReceiver`) that will respond to `SMS_RECEIVED` intents.

Android manifests contain a number of interesting information that can indeed provide preliminary insights about an application maliciousness [73].

4.1.3 Native Interface

While the main technology to develop android application is Java, it is possible to embed small pieces of native code (C, C++), compiled as shared libraries that are dynamically loaded at runtime. Once loaded, native functions can be invoked by Java code and are subject to the same restrictions. Benign applications use native code to perform CPU-bound operations (e.g., a physical engine) that require little interaction with other components. Malicious apps, on the other hand, are known to leverage native code to perform low-level operations such as triggering vulnerabilities to escale privileges or obfuscating the app's code [58]. As an alternative

way to execute native ARM code, an app could include an ELF into its resources and later execute it.

4.1.4 Zygote

As we briefly stated in Section 4.1, every application is sandboxed and is executed by an instance of the Dalvik virtual machine (DVM) [61]. However, cold-starting a new DVM for every application would be too time consuming. To prevent this, Android uses a concept called *Zygote* to enable both sharing of code and fast startup of new instances. The Zygote is started at boot time, it initializes a DVM (that consequently loads every resource it requires) and waits for requests from runtime processes. Upon such requests, the Zygote produces (forks) a new instance of the pre-loaded DVM. By doing so, the DVM startup time is minimized as every resource is duplicated by the fork operation.

4.1.5 Binder: IPC and RPC

The Android system implements the principle of least privilege by providing a sandbox for each installed application. One process must not manipulate the data of another process and can access only the components the system granted the requested permissions for. Nevertheless, often, applications need a way to communicate to each other and share data, e.g., an application can request the permission to send SMS through the appropriate service.

The Android OS and applications strongly rely on interprocess communication (IPC) and remote procedure calls (RPCs). To this end, Android relies on Binder, a custom implementation of the OpenBinder protocol [74]. The Binder protocol is quite complex, therefore in the following we highlight only the information needed to understand CopperDroid's analysis.

Just like any other RPC mechanism, Binder allows a Java process (e.g., an application) to invoke methods of remote objects (e.g., services) as if they were local methods, through synchronous calls. This is transparent to the caller and all the underlying details (e.g., message forwarding to receivers, start or stop of processes) are handled by the Binder protocol during the remote invocation. To work properly, the caller application must know the remotely-callable methods with parameters. When a service needs to provide a binding, it must define a client-server interface that allows applications to bind and interact with it; such an interface is called *bound service*. If the service is used by other applications or across separate processes and requires multithreading, then the interface is usually defined by means of the Android interface definition language (AIDL). Once defined, an AIDL file is used to automatically generate client- and server-side code in the form of a proxy class, used by a caller, and a stub class, extended by the callee to

implement the logic of the service. The AIDL files of core Android services are available online. As described later, CopperDroid relies on such interfaces to *automatically* infer the interactions between applications from low-level events. Although a few AIDL files may be missing (e.g., custom services), CopperDroid has never experienced such an issue in our current experiments and we are nonetheless investigating automatic reverse engineering technique to overcome such an issue. AIDL performs all the work to decompose objects into primitives that can be marshalled across processes. Any kind of request and data exchanged among clients and services go through Binder, whose thorough analysis allows therefore to identify Android-specific behaviors (e.g., sending an SMS and accessing private information).

Specifically, when IPC is performed from process $\mathscr{A}$ to process $\mathscr{B}$, the calling thread in $\mathscr{A}$ will wait until the next available thread in the thread pool of $\mathscr{B}$ replies with the results. The calling thread returns as soon as it receives such results. The data sent in the transaction is a `Parcel`, a buffer of flattened data and meta-data information. Dispatching of the message between $\mathscr{A}$ and $\mathscr{B}$ takes place by means of a `ioctl` system call handled by the Binder kernel driver.

4.2 Related Literature

In this section we cite and compare against the most relevant work we believe directly relates with CopperDroid.

4.2.1 Current Techniques

DroidScope [63] is a framework to create dynamic analysis tools for Android malware that trades off simplicity and efficiency for transparency: as an out-of-the-box approach, it instruments the Android emulator, but it may incur high overhead (for instance, when taint-tracking is enabled). DroidScope leverages VMI (Virtual Machine Introspection) [62] to gather information about the system and exposes hooks and a set of APIs, which enable the development of plugins to perform both fine and coarse-grained analyses (e.g., system call, single instruction tracing, and taint tracking). Differently from CopperDroid, DroidScope just offers a set of hooks that can be used to build analysis to intercept interesting events but does not perform any behavioral analysis *per-se*. For example, a tool leveraging DroidScope can intercept every system call executed on an Android system, but would still need to do its own VMI to inspect the parameters of each call. Following this principle, CopperDroid could have been built on top of DroidScope, but at the time we implemented it, the DroidScope framework was not publicly available. Moreover, the main focus of our research is *not* to illustrate how to build a frame-

work or a clever VMI technique for Android systems, but rather to point out how a proper system call-centric analysis—which includes a thorough IPC and RPC Binder-related protocol analysis—and stimulation technique can comprehensively expose Android malware behaviors, as shown by our extensive evaluation.

Enck *et al.* presents TaintDroid [75], a framework to enable dynamic taint analysis of Android applications. TaintDroid's main goal is to track how sensitive information flow between the system and applications or between applications to automatically identify leaks. Because of the complexity of the Android system, TaintDroid relies on different levels of instrumentation to perform its analyses. For example, to propagate taint information through native methods and IPC, TaintDroid patches JNI call bridges and the Binder IPC library. TaintDroid is both extremely effective, as it allows to propagate tainting between many different levels, and efficient, as it does that with a very low overhead. Unfortunately, the price to pay is low resiliency and transparency: modifying internal components of Android inevitably exposes TaintDroid to a series of detection and evasions techniques. For instance, even applications with standard privileges can detect TaintDroid's presence by calculating checksums over instrumented and readable components. Moreover, TaintDroid cannot track taintedness of native code. Conversely, applications that can escalate their privileges can go even further: identifying and disabling TaintDroid's hooks and analysis. Furthermore, the decision of modifying internal components also exposes TaintDroid to the problems deriving from constantly adapting the analysis code to an highly-mutable architecture, such as the Android OS.

DroidBox is a dynamic Android malware analyzer [76]. While similar in concept, CopperDroid and DroidBox have a main difference: the latter does not perform out-of-the-box analyses but leverages custom instrumentation of the Android system and kernel to track a sample's behavior, relying on TaintDroid to perform taint tracking of sensitive information [75]. Results of the analysis are produced through Android's standard logging mechanism (e.g., `logcat`), augmented to include information about suspicious behaviors. Extended log messages are then parsed offline. Using TaintDroid and instrumenting Android's internal components makes DroidBox prone to the problems of in-the-box analyses: malware can detect and evade the analyses or, worse, even disabling them.

Andrubis [77] is an extension to the Anubis dynamic malware analysis system to analyze Android malware [78, 79]. According to its web site, it is mainly built on top of both TaintDroid [75] and DroidBox [76] and it thus shares their weaknesses (mainly due to operating "into-the-box"). In addition, Andrubis does not perform any stimulation-based analysis, limiting its effectiveness in discovering interesting Android-specific behaviors.

Aurasium [80] is a technique (and a tool) that enables dynamic and fine-grained policy enforcement of Android applications. To intercept relevant events,

Aurasium instruments single applications, rather than adopting system-level hooks. Working at the application level, however, exposes Aurasium to easy detection or evasion attacks by malicious Android applications. For example, regular applications can rely on native code to detect and disable hooks in the global offset table even without privilege escalation exploits. Aurasium's authors state that their approach can prevent such attacks by intercepting `dlopen` invocations needed to load native libraries. It is however unclear how benign and malicious code can be distinguished, as this policy cannot be lightheartedly delegated to Aurasium's end-users. Conversely, CopperDroid's VMI-based system call-centric analysis is resilient to such evasions.

Google Bouncer [81], as its name suggests, is a service that "bounces" malicious applications off from the official Google Play (market). Little is known about it, except that it is a QEMU-based dynamic analysis framework. All the other information come from reverse-engineering attempts [82] and it is thus impossible to compare it against our approach.

SmartDroid [83] leverages a hybrid analyses that statically identify paths that lead to suspicious actions (e.g., accessing sensitive data) and dynamically determine UI elements that take the execution flow down paths identified by the static analysis. To this end, the authors instrument both the Android emulator and Android's internal components to infer which UI elements can trigger suspicious behaviors. They furthermore evaluate SmartDroid on a testbed of 7 different malware samples. Unfortunately, SmartDroid is vulnerable to obfuscation and reflection, which make it hard—if not impossible—to statically determine every possible execution path. Conversely, CopperDroid's dynamic analysis is resilient to static obfuscation and reflection attempts.

Anand *et al.* propose ACTEve [84], an algorithm that leverages concolic execution to automatically generate input events for smartphone applications. ACTEve is fully automatic: it does not require a learning phase (such as capture-and-replay approaches) and uses novel techniques to prevent the path-explosion problem.

We acknowledge that although CopperDroid's stimulations are proper, its approach is a best-effort attempt and could benefit from state-of-the-art techniques. We however must keep in mind the overhead such techniques may introduce. For instance, the aforementioned work (i.e., [83, 84]) do not seem to pay much attention about performance issues. SmartDroid [83] reports no overhead measurements and the average running time of ACTEve as reported in [84] falls within the range of *hours*, which makes it ill-suited to automated large scale analyses.

4.3 CopperDroid

Our goal is to provide to the analyst a transparent environment to *automatically* perform out-of-the-box dynamic behavioral analysis of any kind of Android applications (and, for this work, we are specifically interested in Android malware). To this end, CopperDroid presents a *unified* analysis to characterize low-level OS-specific behaviors (e.g., writing to a file, executing a program) and high-level Android-specific (e.g., accessing personal data, sending an SMS) behaviors.

In particular, based on the observation that such behaviors are all achieved through the invocation of system calls, CopperDroid's VMI-based system call-centric analysis faithfully describes Android malware behavior whether it is initiated from Java, JNI or ELF code.

Android applications rely on IPC mechanism due to both operating system design and the need to interoperate among services. An application that requires, for instance, to send an SMS, must perform a remote method invocation of the corresponding service. Any exchanged message between the client and service takes place via the Binder protocol, which is implemented as a kernel driver. Therefore, when an application needs to perform IPC, it has to invoke the appropriate `ioctl` system call to allow Binder to dispatch the requested action to the corresponding service and viceversa. Android Binder marshalls and unmarshalls the content of the IPC message, i.e. a Parcel object, based on the information provided in the AIDL (see Section 4.1.5).

To this end, we provide and implement in CopperDroid a novel technique to perform automatic unmarshalling of any AIDL available on the system. This allows to easily retrieve a human and error-free representation of the content of the IPC message, which is of paramount importance to describe and understand Android-specific behaviors (e.g., sending an SMS, accessing private information). Android malware low-level OS-related behaviors (e.g., opening and writing to a file, creating a process, sending network data) are of course achieved through system calls and therefore intercepted[1] by CopperDroid unified analysis.

In other words, any representative application behavior is the union of low and high level information identified by system calls, parameters and Binder unmarshalled data. This result highlights and emphasizes the strength of an unified analysis: a mere system call tracking would not provide any behavior insight if were not combined with Binder information.

[1]The CopperDroid emulator intercepts system calls and extracts their parameters—Binder is a specific case of such.

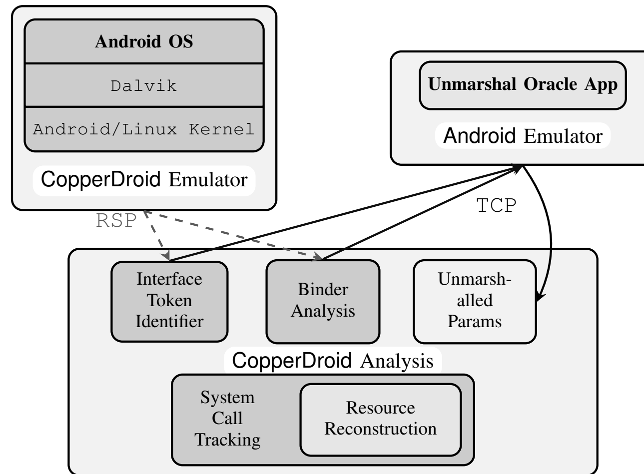


Figure 4.2: CopperDroid Architecture.

4.3.1 CopperDroid Architecture

The architecture of CopperDroid is shown in Figure 4.2.

Our whole Android system runs on top of a modified Android emulator (the CopperDroid emulator), which is built on top of QEMU [64]. To this end, we have enhanced (i.e., instrumented) the Android emulator to enable system call tracking and support our out-of-the-box system call-centric analyses. As Figure 4.2 shows, *all our analyses* are executed outside the CopperDroid emulator and we rely on virtual machine introspection (VMI) [62] to fill the semantic gap between our emulator and the Android OS.

To allow for a flexible host-to-emulator communication and introspection, CopperDroid leverages the remote serial protocol (RSP) of the GNU debugger [85] (see Figure 4.2). The Android emulator provides GDB support via GDB stubs to developers. A GDB stub is an implementation of RSP, which enables the target machine to communicate with the host machine on which a remote GDB session with a client is established. Therefore, any client that is able to communicate over RSP can debug the target machine. Please note that this *does not* modify *any-how* the analyzed Android system, nor it can be detected by apps running inside CopperDroid's emulator.

Relying on RSP allows for an interesting twofold way of analyzing malicious samples. Analysts can in fact opt for manual (through a debugger) or automatic analyses, allowing them to choose between the one that better fits their specific needs of the moment. For instance, a manual analysis may be the best initial choice for a quick-and-dirty examination of an unknown sample, which can eventually drive the analyst to develop a thorough analysis as a CopperDroid's plugin written in the Python programming language.

4.3.2 Processes and Threads

In addition to tracking system calls, CopperDroid provides information about all the processes and threads running on the system. Not only this allows an analyst to maintain an up-to-date view of the system state (e.g., knowing whether a malware spawns other processes or kill existing ones, for instance, as part of a privilege escalation exploit), but it is also of primary importance to allow for a thorough IPC and RPC analysis, as explained in the next section.

As outlined in Section 4.1, each Android application executed on the OS is encapsulated in a Dalvik VM, which runs as a user-space process with its own kernel process descriptor. Such a descriptor is identified by a `task_struct` Linux kernel data structure, which contains important process-related information.

The Linux kernel identifies a process with a process ID (PID), stored at a specific offset within the `task_struct`. Linux associates a distinct PID to each process or lightweight process. A lightweight process is similar to a regular one, but it shares a unique thread group leader and resources with other lightweight processes of a given process. The thread group leader ID (TGID) is also stored at a specific offset within the `task_struct`, which is itself a field of the `thread_info` structure.

To gather information about all the processes and threads running (or ready to run) on the system, CopperDroid must retrieve the address of the current process' `thread_info` variable (through which is possible to retrieve all the other running or ready-to-run threads and processes). Such an address coincides with the bottom of the pages shared with the kernel stack. To retrieve such an address, similarly to the approach adopted by the Linux kernel, CopperDroid masks out the 13 least significant bits of the kernel stack pointer.

It is worth noting that CopperDroid strives to be as transparent as possible, working properly even in the absence of any kernel or debugging symbols. It only relies on the knowledge of a limited number of well-known offsets within the main kernel data structures (e.g., `thread_info`) to retrieve the semantic information mentioned above.

4.3.3 Tracking System Call Invocations

Tracking system call invocations is at the basis of virtually all the dynamic malware behavioral analysis systems [78, 86, 87]. Most—if not all—of such systems implement a form of VMI to track system call invocations on a virtual x86 CPU. Although similar, the ARM architecture underlying the Android emulator—and therefore CopperDroid—presents a few details that may challenge VMI-based system call invocations tracking and are thus worth to rough out.

The ARM ISA provides the `swi` instruction for invoking system calls, which

causes the well-known user-to-kernel transition by triggering a software interrupt. Once the `swi` instruction is executed, the `cpsr` register is set to `supervisor` mode with the program counter register pointing to the system call handler. To track system call invocations, we instrument QEMU when the `swi` instruction is executed. That instruction is not (dynamically) binary translated and can therefore easily be intercepted when QEMU handles the proper software interrupt. To allow trading off completeness for performances, our instrumentation allows to dynamically choose a set of processes and system calls of relevance. When the `swi` instruction is intercepted, we check if a system call is actually being invoked, if that is in the list of the to-be-tracked system calls, and if the current process is in the list of the to-be-monitored processes. If such conditions hold, our CopperDroid-modified emulator raises a debug interrupt which causes the GDB stub to notify the CopperDroid analysis component, running *outside* the instrumented emulator, that a system call is about to start executing. Of course, it is also of paramount importance to detect when a system call is about to return as that allows to save its return value, which enriches the analysis with additional semantic information. Usually, the return address of a system call invocation instruction `swi` is saved in the link register `lr`. While it seems natural to set a breakpoint at that address to retrieve the system call return value, a number of system calls may actually not return at all (e.g., `exit`, `execve`). Therefore, instead of relying on a cumbersome heuristic, the generic approach CopperDroid adopts is to intercept CPU privilege-level transitions.

In particular, CopperDroid detects whenever the `cpsr` register switches from supervisor to user mode (`cpsr_write`), which allows to uniformly retrieve system call return values, if any.

4.3.4 Automatic AIDL Unmarshalling

As outlined in Section 4.1.5, the Android system heavily relies on kernel implemented IPC and RPC channels to carry out tasks and (some) permission-related policy enforcement. Therefore, tracking and dissecting the communications that happen over this media is a key aspect for reconstructing high-level Android-specific behaviors. Although recently explored to enforce user-authorized security policies [80], to the best of our knowledge, CopperDroid is the first approach to carry out a detailed analysis of such communication channels to comprehensively characterize OS-specific and Android-specific behaviors of malicious Android applications.

Let us consider an application that sends an SMS as our running example. From a high-level perspective (e.g., Java methods), sending an SMS roughly corresponds to obtaining a reference to an instance of the class `SmsManager`, the phone SMS manager, and sending the SMS out by invoking the method

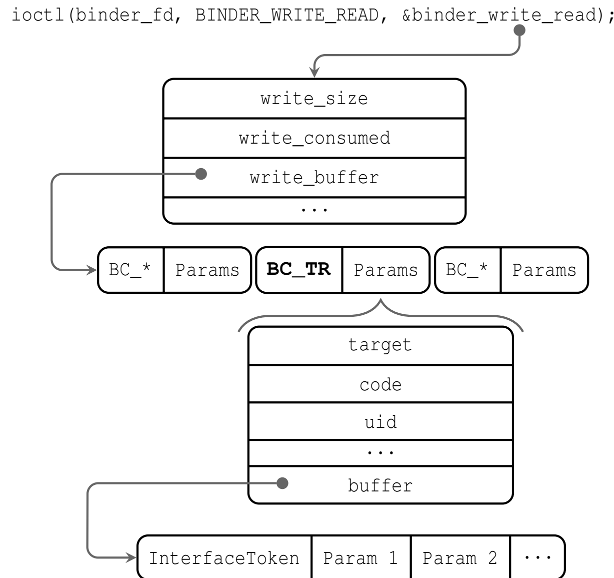


Figure 4.3: Parameters of a `BINDER_WRITE_READ` `ioctl`.

`sendTextMessage` on the instance, with the destination phone number and the text message as the method's arguments. This corresponds to locating the Binder service `isms` and remotely invoking its `sendText` method with proper arguments.

Conversely, from a low-level perspective, the same actions correspond to the sender application invoking two `ioctl` system calls on `/dev/binder`: one to locate the service and the other to invoke its method. CopperDroid thoroughly introspects the arguments of each binder-related `ioctl` system call to reconstruct the remote invocation. This allows to identify the invoked method and its parameters, enabling *de-facto* to infer the high-level semantic of the operation. Although the Binder protocol implements other `ioctl`s, the `BINDER_WRITE_READ` is the most important one as it allows to transfer data between processes. Figure 4.3 depicts a few details about the parameter of these `ioctl`s. As can be observed, they may embed one or more operations for the Binder protocol. These operations are stored sequentially in the `write_buffer` field of the `ioctl`'s last argument.

In particular, we focus our analysis on Binder *transactions*, i.e., IPC operations that actually transfer data (also responsible for RPC). To identify them, CopperDroid parses the memory structures passed as a parameter to the `ioctl` system call and identifies `BC_TRANSACTION` and `BC_REPLY` (see [65] for further details).

However, just intercepting transactions may be of limited use when it comes to understand Android-specific behaviors. In fact, the Binder `ioctl`-provided raw data that flow throughout transactions are in the form of `Parcel` (marshalled) objects. Moreover, every interface the client and service both agree upon has its own set of predefined methods signature. As the Android framework counts about

Listing 4.1: Oracle Android Application

```
Parcel unparcel = Parcel.obtain();
// 1) Receive byte array ''data'' and string list ''types''
RunTCPSocket();

for ( String type : types) {
 unparcel.unmarshall(data, offset, data.length);
 // 2a) Unmarshall Primitives and Primitive Arrays
 if (''double''.equals(type)) {unparcel.readDouble();}
 if (''string''.equals(type)) {unparcel.readString(); }
 if (''BooleanArray''.equals(type)) {
  Arrays.toString(parcel.createBooleanArray()); }

 // 2b) Unmarshall Class Objects
 Class cl = Class.forName(type);
 Parcelable g=unparcel.readParcelable(cl.getClassLoader());
 Field CT = cl.getDeclaredField(''CREATOR'');
 Creator creator = (Creator) CT.get(g);
 Object unparceled = creator.createFromParcel(unparcel);

 // Update string list ''out'' and ''offset''
}

// 3) Send Unmarshalled String Representations
runTcpSend(out);
```

300 AIDL interfaces, manual unmarshalling is unfeasible.

To understand our novel proposed automatic unmarshalling technique consider the relationship between the unmarshaller and CopperDroid in Figure 4.2.

CopperDroid first acquires, at run-time, IPC/RPC binder related data from the leftmost emulator, then redirects the data to our constructed Oracle application residing in the rightmost emulator. It is important to note that, while the leftmost emulator has been altered to fit CopperDroid's needs, the emulator running the Oracle is a separate, unaltered Android emulator preventing local privileged malware from altering communications from CopperDroid to hide itself. This redirection is split into two sets of data of 1) marshalled data derived from binder communication by the CopperDroid "Binder Analysis" block and 2) a matching list of primitive types and class names corresponding to the sequence of marshalled data, created through the utilization of intercepted AIDL tokens, or identifiers. Once acquired, and with the use of Java reflection, the Oracle is able to unmarshal all the complex serialized Java objects, returning all string representations of unmarshalled data to CopperDroid for further Android-specific behavioral analysis.

In detail, the Oracle Android application may unmarshal the binder communication in two unique ways, depending on whether the type of data is a primitive type or a class object. White iterating through the list of types and class names, if the type is identified as primitive, the correct read function provided by `Parcel` is implemented. Part 2a of Listing 4.1 shows an example of reading both primitive and primitive data arrays from a parcel. Alternatively, for unmarshalling class objects, `Parcel` provides `Parcelable` methods that allow objects to both write and read themselves into `Parcels`. Listing 4.1 part 2b depicts the unmarshalling of an `Intent` class object; including both the class type and the class data.

Unlike primitives, to unmarshal a class instance, the Oracle application requires Java reflection [88] and the "creator" type before reading the remaining class data: in our example, the class data of an Android `Intent` entails a class name, action, and extras. The creator field is similar to the "type" already provided by CopperDroid, but is only implemented when unmarshalling class objects. Moreover, the creator field must be readable if the Oracle is to correctly construct the new object and propagate the new object with the remaining class data. We can see this in Listing 4.1, where the type "Intent" is used in Oracle step 2b to manipulate parcelable protocols and acquire that class object's `Parcelable.Creator`, and then the creator is properly used construct the new object and read in its data.

Once either a primitive or class type has been unmarshalled, the string representation is appended to an output string list, and the marshalled data offset is updated to point of the next unmarshalled item. Additionally, the Oracle iterates to the next type or class name on the given list. This can be seen in the for loop encasing Listing 4.1 step 2 in our Oracle. This automated multi-threaded Oracle relieves us of the daunting manual effort of unmarshalling all 300 possible AIDL interfaces, and handles multiple large requests both quickly and accurately.

4.3.5 Resource Reconstructor

The purpose of our resource reconstructor [89] is twofold: to map a stream of related low-level events to a more meaningful high-level behavior, and to recreate any file a sample software may have created.

By implementing data dependencies within a stream of low-level system calls, including their parameters, CopperDroid is capable of abstracting high-level behaviors such as file accesses. We capture these notions of behavior by first intercepting all system calls between the candidate application in the left emulator of Figure 4.2 and the system. A sample stream of system calls, as seen by the analysis block in Figure 4.2, is provided for consideration in Listing 4.2. In this example, initial analysis by the CopperDroid "System Call Tracking" block is capable of detecting a file access action from the highlighted lines, however, with the addition of forward slicing and data dependencies in the embedded reconstructor, not only will the file access be detected, but the file "tasks" will be recreated in the sdcard directory with the value "0" written to it.

More specifically, as the reconstructor performs forward slicing on the stack of system calls, it selects the set of instructions associated with each file opening and re-creates their effects. As each system call is emulated, meta data such as flags, process and group process IDs, are retained as the contents of the recreated file are edited accordingly. This is essential as each set of system calls, including instructions such as open/close (fd, flags, and mode), write/writev (new text, fd, fd's mode, fd's offset), lseek (fd, fd's offset), dup (old fd, new fd, flags, mode,

Listing 4.2: Trace File will System Calls and Parameters

```
[c5b02000-35-35-zygote] fork( ) = 0x125
[c5b02000-35-35-zygote] getpgid( 0x41 ) = 0x23
[c5b02000-35-35-zygote] setpgid( 0x125, 0x23 ) = 0x0
[c1c18000-293-293-zygote] getuid32( ) = 0x0
[c1c18000-293-293-zygote] open(/sdcard/tasks, 0x20242, 0x1b6) = 0x13
[c1c18000-293-293-zygote] fstat64( 0x13, 0xbef7f910 ) = 0x0
[c1c18000-293-293-zygote] mprotect( 0x40008000, 0x1000, 0x3 ) = 0x0
[c1c18000-293-293-zygote] mprotect( 0x40008000, 0x1000, 0x1 ) = 0x0
[c1c18000-293-293-zygote] write( 0x13 - /sdcard/tasks, 0xa24c0 ‘0’, 0x1 ) = 0x1
[c1c18000-293-293-zygote] close( 0x13 ) = 0x0
[c1c18000-293-293-zygote] prctl( 0x8, 0x1, 0x0, 0x0, 0x0 ) = 0x0
[c1c18000-293-293-zygote] setgroups32( 0x2, 0xbef7fa20 ) = 0x0
[c1c18000-293-293-zygote] setgid32( 0x2722 ) = 0x0
```

new offset), and unlink (fd)[1], must share some of the same meta data as depicted by data dependencies (e.g. file descriptors).

This process of automatically grouping related sequences of low-level system calls allows us to abstract more meaningful high-level behaviors, such as file recreation. Using real world samples we were able condense large traces into high-level behaviors, recreate malicious files such as rageagainstthecage [90] or other applications, which can be fed back into CopperDroid to uncover additional exhibited behaviors and its intended use.

4.3.6 Path Coverage

Although effective, a simple install-then-execute dynamic analysis may miss a number of interesting (malicious) behaviors. On the one hand, this problem has long been affecting traditional dynamic analysis approaches as non-exercised paths are simply not analyzed. If such paths host additional (or the only) malicious behaviors, then any dynamic analysis would fail unless proper, but generally expensive and complex exploration techniques are adopted [91, 92]. On the other hand, this problem is exacerbated by the fact that mobile applications are inherently user driven and interaction with applications is generally necessary to increase coverage. For instance, let us consider an application with a manifest similar to the one depicted in Figure 4.1. After installation, the application would only react to the reception of SMS, showing no interesting nor additional behaviors otherwise.

Traditional executables have a single entry point, while Android applications may have multiple ones. Most applications have a main activity, but ancillary activities may be triggered by the system or by other applications and the execu-

[1] Unlink is not fully recreated, as that would remove files we are interested in. However, the attempt itself is an action worth noting, therefore files are simply renamed to reflect this system call.

#	Stimulation Type	Parameters	Cond.
1	Received SMS	*Text, from number*	✓
2	Incoming call	*From number, duration*	✓
3	Tapping	*Coordinates, pause*	
4	Location update	*Geospatial coordinates*	✓
5	Battery status	*Amount of battery*	✓
6	Keyboard input	*Typed text*	
7	Phone Reboot	-	✓

Table 4.1: Main stimulations and parameters. A ✓ identifies a conditional stimulation.

tion may reach them *without* flowing through the main. To address such coverage problem, CopperDroid implements a novel approach (based on extracting information from the malware Manifest) to artificially stimulate the analyzed malware with a number of events of interest. For example, injecting events such as phone calls and reception of SMS texts would lead to the execution of the registered application's broadcast receivers. Another example that comes from our experience with Android Malware is the `BOOT_RECEIVED` intent, that many samples use to get executed as soon as the victim system is booted (much like `\CurrentVersion\Run` registry keys on Windows systems).

The Android emulator offers the possibility to inject a considerable number of artificial events to stimulate a running application. These range from very low-level hardware-related events (e.g., loss of the 3G signal) to higher-level ones (e.g., incoming calls, SMS). CopperDroid could adopt a fuzzing-like stimulation strategy and trigger *all* the events that could be of interest for the analyses, ignoring information that can be extracted from the target application. That would unfortunately be of limited effect because of the underlying Android security model and permission system, which can instead be leveraged to carry out a fine-grained targeted stimulation strategy. To this end, CopperDroid examines applications manifest to extract events and permission-related information to drive the malware stimulation approach.

There are a few exceptions to the aforementioned must-declare-everything rule. Tapping and keyboard interactions are implicit and allowed to every Android application. Therefore, a limited number of stimulation are always performed, regardless whether they are declared in the application manifest. Furthermore, an application could *dynamically* register a broadcast receiver for custom events at run-time. CopperDroid is able to intercept such operations and add a proper stimulation for the newly registered receiver.

To perform its custom stimulation, CopperDroid leverages the Android emulator capabilities to inject a number of artificial events into the emulated system. In particular, CopperDroid leverages Monkeyrunner, a tool that provides an out-of-the-box API to control an Android device or emulator, through the Python

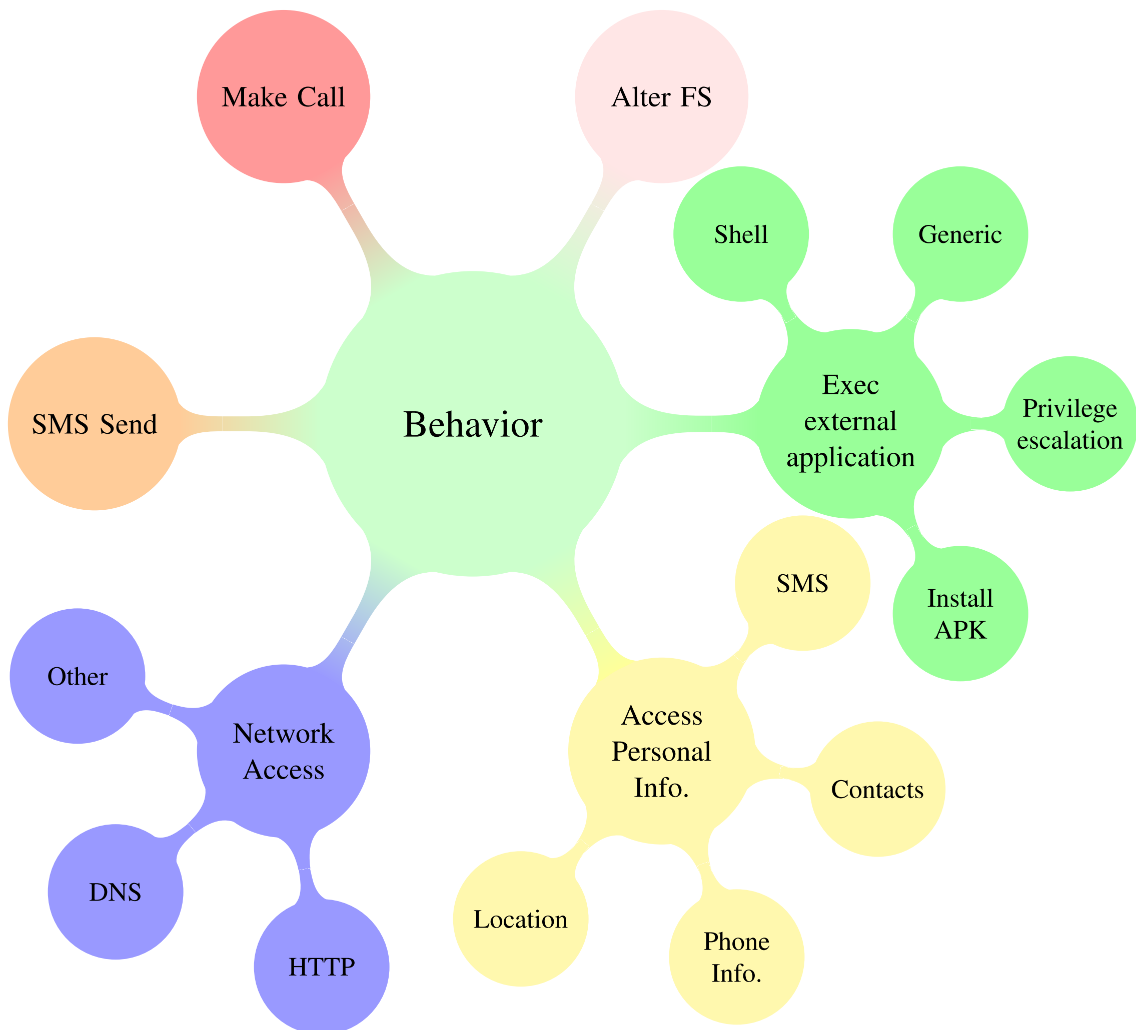


Figure 4.4: Hierarchical map of behaviors.

programming language [93].

A summary of the main events CopperDroid handles is reported in Table 4.1, which also shows the parameters that can be customized for each event. The last column of the table points out whether the stimulation depends on the manifest-extraction mechanism or dynamic triggering.

4.3.7 Suspicious Behaviors

Up to here, we have not yet defined what we consider to be a suspicious behavior[2]. To this end, we manually examined the results of CopperDroid's analyses (i.e., system call invocations tracking and Binder analysis) on a number of randomly selected Android malware extracted from the samples sets at our disposal [67,68].

Figure 4.4 shows the insights of our examination, which allowed us to identify six macro class of suspicious behaviors. Each class contains one or more behavioral model, which is defined by a set of *actions*. Actions are traced through CopperDroid and can belong to any level of behavior abstraction (e.g., OS-specific and Android-specific behaviors).

Interestingly, some behaviors are well-known and shared with the world of non-mobile malware. Others, such as those under the "Accessing Personal Info." class, are instead inherently specific to the mobile ecosystem.

Every terminating class in the map corresponds to a behavioral model that can be expressed by an arbitrary number of actions, depending on its specific complexity. The complexity of these elements is very variable. Some are defined as a single system call, such as `execve`. Others, such as "SMS Send" or those under "Access Personal Info", are defined as a set of transactions of the Binder protocol. Yet others are defined as multiple consecutive system calls. For instance, outgoing HTTP traffic is modeled as a graph with a `connect` system call, followed by an arbitrary number of `send`-like system calls, whose payload is parsed to detect HTTP messages, possibly interleaved by a number of arbitrary unrelated non-socket system calls.

Terminating classes do not forcibly correspond to just one of the aforementioned models but may also contain a set of them. To clarify, consider the following two examples:

```
execve('pm',['pm', 'install', '-r', 'New.apk'],...);
---
Intent intent = new Intent(Intent.ACTION_VIEW);
intent.setDataAndType(Uri.fromFile( \
  new File("/mnt/sdcard/New.apk")), \
"application/vnd.android.package-archive");
startActivity(intent);
```

CopperDroid recognizes actions triggered by both these snippets of code as belonging to the class "Install APK", but yet they are very diverse (respectively, a system call and a Binder protocol transaction).

The behavioral map in Figure 4.4 has been built on top of the experiments conducted on a large corpus of malware, but we are well-aware that it is very hard, if not impossible, for it to be complete.

[2]It is worth noting that CopperDroid is able to automatically reconstruct the whole behavior of an Android malware, being it suspicious or not.

Currently we do not perform malware detection. Nevertheless, CopperDroid tries to address the semantic gap problem in behavioral monitoring by classifying actions into high behavioral classes. To achieve this, we observe temporal sequence of system calls, their parameters and the use of system resources. Based on these information we are able to abstract OS-specific and Android-specific behaviors into high-level behaviors, identified with classes. At the time of writing, we rely on our custom regular expression behavioral patterns and detection engine to identify behavioral models. Figures 4.5 and 4.6 show examples of reports provided by our CopperDroid web service.

Host

ID	CLASS	SUBCLASS	TID	PROC
0	FS ACCESS - [appA.db]		319	processA
1	ACCESS PERSONAL INFO	SMS	117	Binder Thread #
2	ACCESS PERSONAL INFO	ACCOUNT	345	a.process.core
3	SMS SEND		319	processA

TS	TYPE	METHOD
2013-11-11 08:49:40	BINDER	com.android.internal.telephony.ISms.sendText(destAddr = 0000, scAddr = None, text = From:0011223344:Here we are!, sentIntent = [PendingIntent N/A], deliveryIntent = [PendingIntent N/A])

ID	CLASS	SUBCLASS	TID	PROC
4	ACCESS PERSONAL INFO	SMS	117	Binder Thread #
5	SMS SEND		319	processA
6	ACCESS PERSONAL INFO	ACCOUNT	353	Binder Thread #

Figure 4.5: Application Trace: Binder information and unmarshalled data

ID	CLASS	SUBCLASS	TID	PROC
21	FS ACCESS - [com.test-1.dex]		347	dexopt
22	EXECUTE		347	com.test
23	EXECUTE	SHELL	353	com.test

TS	TYPE	NAME	ADDITIONAL INFO
2013-08-17 19:13:41	SYSCALL	execve	{'executed_file': '/system/bin/sh', 'args': " ['/system/bin/sh', '-']", 'retval': 0}

ID	CLASS	SUBCLASS	TID	PROC
24	FS ACCESS - [310.xml]		319	com.test

TS	TYPE	NAME	ADDITIONAL INFO
2013-08-17 19:13:48	SYSCALL	open	{'flags': 131649, 'mode': 384, 'filename': u'/data/data/com.test/shared_prefs/310.xml \x00'}
2013-08-17 19:13:48	SYSCALL	write	{'filename': u'/data/data/com.test/shared_prefs/310.xml '}

Figure 4.6: Application Trace: Command Execution and File System Access

Defining and formalizing additional behaviors as well as adopting a more sophisticated and resilient abstract behavior language [94,95] are part of our ongoing research effort.

Criterion	Gdl.	Respected	Criterion	Gdl.	Respected
Removed Goodware	A.1	✓	Described NAT	B.5	✗
Balanced Families	A.2	✗	Interpreted FPs/FNs	B.6	✗
Separated Datasets	A.3	✗	Interpreted TPs	B.7	✗
Higher Privileges	A.4	✓	Removed moot samples	C.1	✓
Mitigated Artifacts	A.5	✓	Used Many Families	C.1	✓
Avoided Overlays	A.6	✓	Real-world FPs/TPs exp.	C.2	✗
Listed Malware	B.1	✓	Used multiple OSes	C.3	✗
Listed Malware Families	B.2	✓	Added user interaction	C.4	✓
Described Sampling	B.3	✗	Allowed Internet	C.5	✓
Mentioned OS	B.4	✓			

Table 4.2: Prudent guidelines, as defined in [96].

4.4 Evaluation

Our experimental setup is as follows. We ran an unmodified Android Gingerbread image[3] on top of our CopperDroid-enhanced emulator. The system was customized to include personal information, such as contacts, SMS texts, call logs, and pictures to mimic as closely as possible a real device. Each analyzed malware sample was installed in the emulator and traced until a timeout was reached. At the end of the analysis, a *clean* execution environment was restored to prevent corruptions and side-effects caused by installing more than one malware sample in the same system. To limit noisy results, each sample was executed and analyzed 6 times: thrice *without* stimulation and thrice *with* stimulation; results of single executions were then merged.

As shown in Table 4.2, our experiments were designed to comply as much as possible with recently presented guidelines when performing experiments on malware [96]. Please note that most of the unmet principles were out of scope to CopperDroid (e.g., no FPs nor TPs are reported as currently CopperDroid does not perform any classification nor detection), while a few others simply required additional time (e.g., C.3). However, contrarily to related work (e.g., DroidScope [63] and Aurasium [80]), CopperDroid is *independent* on the underlying Android system it analyzes (for instance, we have successfully ported CopperDroid to the latest Android version, already). Part of our future plans is to analyze our data set in such new settings to explore whether and how the behavior of Android malware is influenced by different OS versions.

We performed a threefold experiment on three different malware datasets: two publicly available [67, 68] and one provided by McAfee [97], respectively com-

[3] As of March 2013, Gingerbread is still the most widespread Android version.

Malware Dataset	Samples w/ Add. Behaviors	Avg. Increment	Std. Dev
Genome	752/1226 (60%)	2.9/10.3 (28.1%)	2.4/11.8
Contagio	289/395 (73%)	5.2/23.6 (22.0%)	3.3/19.8
McAfee UK	836/1365 (61%)	6.5/22.8 (28.5%)	9.5/30.1

Table 4.3: Summary of stimulation results, per dataset.

Malware Family	Samples w/ Add. Behaviors	Behavior w/o Stim.	Incr. Behavior w/ Stim.
LVedu	33/56	26.93	5.2 (19%)
PJApps	36/39	27.41	6.1 (22%)
BaseBridge	10/12	4.5	3.3 (73%)
SMSTrack	4/4	33.5	60.5 (180.6%)
Foncy	2/2	1	4 (400%)

Table 4.4: Excerpt from the overall McAfee samples analysis.

posed of 1,226, 395 and 1,365 samples, counting more than 2,900 samples.

To evaluate the effectiveness of CopperDroid stimulation approach we proceeded as follows. First, we analyzed all the samples without external stimulation. Then, we performed the stimulation-driven analysis of the same malware sets, as outlined in Section 4.3.6.

A summary of the effects of the stimulation on the three datasets is presented in Table 4.3. The results of our analysis on the new McAfee dataset[4](in Table 4.6) shows 836 out of 1365 (61%) McAfee samples exhibited additional suspicious behaviors (see Section 4.3.7 for what we consider to be a suspicious behavior) and, on average, the number of additional behaviors was roughly 6.5, out of an average number of exhibited behaviors of 22.8, observed during non-stimulated executions.

Table 4.4 reports an excerpt of the results of CopperDroid analysis on the McAfee dataset (the complete results are reported in Table 4.6). To exemplify, let us consider the second row. The malware family *PJApps* contains **39** samples of which **36** exhibited additional behaviors when stimulated by the stimulation-driven CopperDroid analysis. More precisely, during the *non*-stimulated executions, we observed an average of **27.41** behaviors for each sample of the family, while the stimulated executions allowed to discover an average of **6.1** additional, *previously unseen*, behaviors.

During the analysis of the McAfee dataset, more than 10% of the samples did not exhibit any behavior, regardless of the stimulation technique adopted. Nearly

[4]We would like to point out that the dataset provided by [67, 68] are 1-year old, while the McAfee dataset evaluated for this dissertation was snapshotted on Feb 2013.

Behavior Class	Stimulation: ✗	Stimulation: ✓
FS Access	889/1365 (65.13%)	912/1365 (66.81%)
Access Personal Info.	558/1365 (40.88%)	903/1365 (66.15%)
Network Access	457/1365 (33.48%)	461/1365 (33.77%)
Exec. External Appf.	171/1365 (12.52%)	171/1365 (12.52%)
Send SMS	38/1365 (2.78%)	42/1365 (3.08%)
Make/Alter Call	1/1365 (0.07%)	55/1365 (4.03%)

Table 4.5: Overall behavior breakdown of McAfee samples.

half of these samples did so because they could not get installed successfully on the CopperDroid-enhanced emulator, while the other half stayed likely dormant or did not exhibit any interesting behavior until CopperDroid analysis timeout hit (due to code coverage issue or VM evasions, for instance). Reasons are manifold, and investigating such issues is part of our ongoing research effort. However, it is worth noting that such limitations are not specific to CopperDroid, but are open issues of dynamic analysis or VM-based techniques [98], in general.

As we discussed in Section 4.2, solutions to improve code coverage may be built on top of symbolic execution [84, 99], for instance, but unfortunately they do not scale well and are ill-suited to perform large scale analyses such as those performed by CopperDroid.

Table 4.5 reports the overall breakdown of the observed behaviors (see Figure 4.4) on McAfee dataset. Each row identifies the class of behavior and how many samples over the total exhibited at least one occurrence of such behavior, *without* and *with* stimulation, respectively. As can be observed the two most influenced behavioral class are *Access Personal Information* and *Make/Alter Call*. The first is triggered by a non-negligible number of samples that receive an incoming message sent by CopperDroid stimulation technique (and exhibit an access to the user's personal information, otherwise hidden). The latter is mostly due to a set of malware that, whenever a call is received, hide its notification to the user.

4.4.1 Performance Evaluation

In this section we present an evaluation of CopperDroid's introduced overhead obtained through a number of different experiments conducted on a GNU/Linux Debian 64-bit system equipped with an Intel 3.30GHz core (i5) and 3GB of RAM. Benchmarking a multi-layered system, such as Android, in conjunction with a complex technique, such as CopperDroid (and in an emulated environment), can be a rather complicated task. For example, traditional benchmarking suites based on measuring I/O operations are affected by caching mechanisms of emulated

environments. On the other hand, CPU-intensive benchmarks are meaningless with respect to the overhead introduced by CopperDroid, as it mainly operates on system calls.

To address such issues, we performed two different benchmarking experiments. The first is a *macrobenchmark* that tests the overhead introduced by CopperDroid on common Android-specific actions, such as accessing contacts and sending SMS texts. Because such actions are performed via the Binder protocol, these tests give a good evaluation of the overhead caused by CopperDroid's Binder analysis infrastructure. The second set of experiments is a *microbenchmark* that measures the computational time CopperDroid needs to analyze a subset of interesting system calls.

To execute the first set of benchmark, we created a fictional Android application that performs generic tasks, such as sending (`SEND_SMS`) and reading (`SMS`) of SMS texts, accessing local account information (`GET_ACC`), and reading all contacts (`CONTACTS`). We then ran the test application a sufficient number of iterations (i.e., 100 times) and collected the average time required to perform these operations under 3 different settings: on an unmodified Android emulator (i.e., without CopperDroid—baseline), on a CopperDroid-enhanced emulator with CopperDroid configured to monitor the test application (the common setting when analyzing a piece of malware—CD (targeted)), and on a CopperDroid-enhanced emulator with CopperDroid configured to track system-wide events (CD (full)). Results are reported in Figure 4.7 (A). As can be observed, the overhead introduced by the targeted analysis is relatively low, respectively $\approx 26\%$, $\approx 32\%$, $\approx 24\%$ and $\approx 20\%$. On the other hand, system-wide analyses increase the overhead considerably (>2x) because of the number of Android components that are concurrently analyzed.

The second set of experiments measures the average time required to inspect a subset of interesting system calls analyzed by CopperDroid. This experiment collected more than 150,000 system calls obtained by instructing the system to run applications subjected to arbitrary (and artificial) workloads. As tracking a system call requires to intercept entry and exit execution points, we report such measures separately, as shown in Figure 4.7 (B) (the average times are $0.092ms$ for entry and $0.091ms$ for exit).

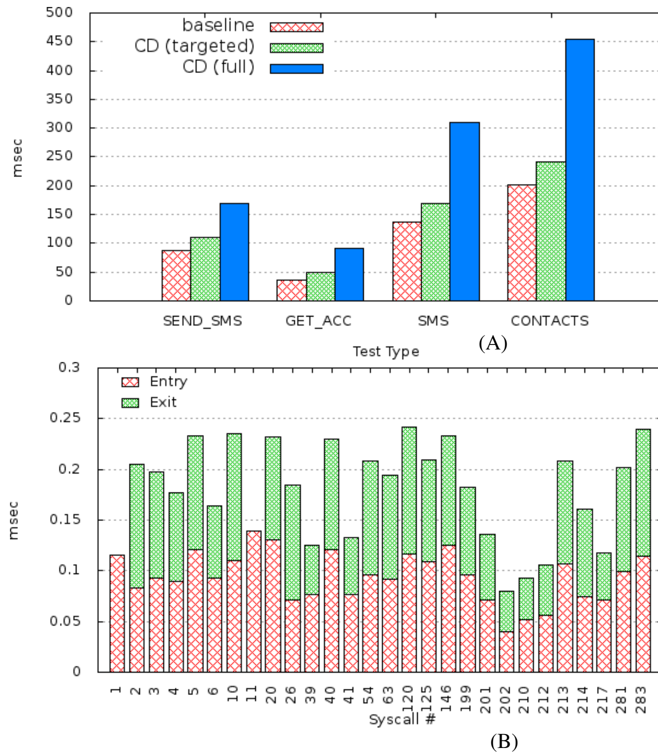


Figure 4.7: Binder Macrobenchmark (A) and System Call Microbenchmark (B).

Malware Family	Samples w/ Add. Behaviors	Behavior w/o Stim.	Incr. Behavior w/ Stim.
Ackposts	1/1	4	3(75%)
Actrack	1/1	4	1(25%)
AndroidSMS	2/2	0	1(⊥)
Anserver	13/21	16.48	5.2(32%)
ApkMon	1/2	49	1(2%)
AppHnd	4/4	37.25	16.8(45%)
AreSpy	1/1	11	6(55%)
Arspam	1/1	3	2(67%)
BackReg	1/1	78	12(15%)
Backscript	2/6	9.67	19.5(202%)
BaseBridge	10/12	4.5	3.3(73%)
Bgyoulu	3/5	17.6	4(23%)
BookFri	1/1	15	4(27%)
Carotap	2/2	4	3(75%)
Coolpaperleek	1/1	55	4(7%)
Crusewin	4/4	6.25	8.5(136%)
Dialer	0/1	1	0(0%)
DiutesEx	23/43	26.58	8.9(33%)
DIYAds	18/18	163.72	37.6(23%)
DougaLeaker	16/16	4	1.6(40%)

Drad	5/5	10.6	6(57%)
Drd.*	30/32	24.74	7.55(31%)
DroidDeluxe	1/1	9	1(11%)
DroidKungFu	63/85	31.02	6.1(20%)
DropDialer	2/11	0	1.5(⊥)
Ecobatry	1/1	25	1(4%)
EICAR	0/2	1.5	0(0%)
Enesoluty	1/1	11	2(18%)
EvoRoot	0/1	0	0(⊥)
Fake.*	314/677	6.39	5.69(89%)
Fladstep	1/1	176	80(45%)
FlashRec	1/2	8	3(38%)
FndNCll	1/1	36	2(6%)
Foncy	2/2	1	4(400%)
FoncyDropper	1/1	23	1(4%)
FrictSpy	8/9	7.56	10(132%)
Frogonal	2/2	27.5	2.5(9%)
Frutk	1/1	73	17(23%)
FunsBot	2/2	5	2(40%)
Gamex	1/1	11	2(18%)
GamexDropper	1/1	8	1(13%)
Geinimi	11/19	23.68	12.4(52%)
GGeeGame	1/1	62	6(10%)
GoldDream	7/8	31.12	9.9(32%)
GoldenEagle	1/1	0	7(⊥)
GoneSixty	11/11	16.64	5.5(33%)
GpsNake	0/1	1	0(0%)
HippoSMS	1/1	16	4(25%)
Hnway	0/1	49	0(0%)
Imlog	5/6	19	9.2(48%)
IMWebViewer	1/1	94	11(12%)
InstBBridge	0/1	9	0(0%)
J	7/13	30.96	3.65(12%)
Jifake	1/5	1	4(400%)
Jmsonez	2/2	11.5	12(104%)
LdBolt	8/8	46.62	7.8(17%)
LoggerKid	4/4	4.5	2(44%)
Logkare	0/1	0	0(⊥)
LoveTrp	1/1	5	6(120%)
LVedu	33/56	26.93	5.2(19%)
Maistealer	1/1	8	1(13%)
Malebook	1/1	94	14(15%)
Mania	1/2	0.5	2(400%)
MarketPay	1/1	98	7(7%)
Mob.*	11/11	43.67	9.75(22%)
Moghava	1/1	0	2(⊥)
MoneyFone	1/1	0	3(⊥)
Nandrobox	1/1	0	4(⊥)
Netisend	1/1	8	4(50%)

NickiSpy	2/2	71	10.5(15%)
NotCompatible	0/1	7	0(0%)
Nyearleaker	1/1	23	5(22%)
OneClickFraud	22/22	16.27	17.2(106%)
PdaSpy	1/4	0	1(⊥)
PJApps	36/39	27.41	6.1(22%)
Qicsomos	0/1	15	0(0%)
QieTing	1/1	0	4(⊥)
QuoteDoor	0/1	6	0(0%)
RecCaller	1/1	2	4(200%)
RootSmart	2/2	17	9(53%)
RuFraud	4/6	4.5	5(111%)
SGSpy	1/1	60	39(65%)
SGSpyAct	0/1	0	0(⊥)
ShdBreak	0/1	28	0(0%)
SilentWap	3/3	2	5(250%)
SMS.*	16/21	4.77	8.59(180%)
Sngo	1/1	65	2(3%)
Spitmo	2/2	0	9(⊥)
SpyBubb	2/2	25.5	20(78%)
Spytrack	1/1	20	8(40%)
Stamper	1/1	63	7(11%)
SteamyScr	2/2	25.5	8.5(33%)
Steek	15/15	8.4	2.1(25%)
Stiniter	0/1	3	0(0%)
Sumzand	0/3	7	0(0%)
SusetupTool	0/1	0	0(⊥)
Sxjspy	1/1	24	4(17%)
TattoHack	1/2	6	1(17%)
Tcent	1/1	0	17(⊥)
ToorKing	1/1	37	6(16%)
ToorSatp	3/8	7.5	1.3(17%)
Toplank	6/9	37.44	6(16%)
Twikabot	1/1	0	12(⊥)
TypStu	4/6	0.83	1(120%)
UranaiCall	1/1	51	13(25%)
VDLoader	10/10	43.7	8.8(20%)
Vidro	1/1	58	16(28%)
Voldbrk	9/17	48.82	1.2(2%)
WalkTxt	1/1	14	2(14%)
Wapaxy	2/2	0	9(⊥)
Woobooleaker	1/1	5	2(40%)
XanitreSpy	9/9	27.11	5.9(22%)
XobSms	1/1	28	15(54%)
YiCha	10/10	21.5	4.6(21%)
Zitmo	3/3	2.67	5.7(213%)

Overall	**836/1365**	**22.78**	**6.54(28.7%)**

Table 4.6: Details of the stimulation results.

5 On the Privacy of Real-World Friend-Finder Services

CopperDroid does not only address malware analysis, but allows to automatically perform out-of-the-box dynamic analysis of any kind of Android applications. To highlight the advantages of such a solution, we present the analysis of a location aware mobile application as a case-study. The analysis results show that even a benign application can lead to privacy leakage when the involved sensitive information are not subjected to any sort of protection to provide privacy data retention. In this context, the malicious author will not force the user to install any malware, instead leverages application privacy leakage to precisely identify the location of the users using this service.

5.1 Background

Friend finders are popular services that allow a user to discover, through her mobile device, people that are *in the vicinity*. We classify these services into four groups, based on two main technical dimension. First, some friend finders allow each user to know the position of other users, for example showing them on a map. We name these services "explicit position" friend finders[1]. In contrast, "implicit position" friend finders only show location-related information, without providing users' *precise* position[2]. For instance, these services only show users closer than a given distance threshold. The second distinguishing technicality is how users are related to each other: in "closed buddies" services, a user is informed about the position (or related information) of other users in a list of "friends", that must

[1]Examples are "Find my friend", and "Google Latitude".
[2]Examples are *PCube*, *SKOUT*, and *Badoo*.

explicitly and *mutually* confirm their willingness to be in such a list. For example, when using *Google Latitude*, a user can define the set of other users who are allowed to see her position on the map. Vice versa, in a "open buddies" approach, all users are considered as "friends". For example, *SKOUT* provides a user with the distances from all nearby users.

When using "explicit position" friend finder services, users are well-aware that their position is being publicly disclosed to other users of the service. In contrast, a user of a "implicit position" service would expect her position to be protected from free disclosure to other users. In some "closed buddies" services this is actually the case. For example in *PCube* users have a fine control over the information they disclose to their friends [100]. Overall, most of the scientific contributions addressing this problem consider "closed buddies" services [101–104].

In this work we consider commercial friend finder services that are "open buddies" and "implicit position". We show that these services provide a deceitful form of privacy protection. Indeed, while a user's position is not directly transmitted to other users, it is possible to compute the position by elaborating the information that the service provider publicly discloses to any user. In particular, we consider one of the most popular dating services that uses a friend finder as one of its functionality. The service declares to have more than a hundred of millions of users in total. Since the attacks that we describe may endanger the privacy of the users of this service, we will not report its name but will only refer to it as "the Service".

To perform such attack, we conducted the analysis of the "the Service" by means of CopperDroid. However, no explicit results and report are provided in order to do not leak details of the analyzed service that could lead to the indentification of the commercial application. Nontheless, this attack show how CopperDroid can be employed for analyzing also benign applications, and that also those that are considered benign can lead to privacy leakage when the involved sensitive information are not subjected to any sort of protection to provide privacy data retention.

To summarize, we provide the following three main contributions.

1) We describe two different attacks to obtain the position of any user and we give an example of how to perform the attacks manually, i.e., without any ad-hoc software (see Section 5.2).

2) We show how the attacks can be fully automated, through the use of an *ad-hoc* client that can compute, in a few seconds, the position of any user in a given area (see Section 5.3).

3) We describe how the identification of the position of a user in a given area can be used as a primitive to develop even more threatening attacks (see Section 5.4).

5.2 Attack description

In this section we first clarify our reference scenario (Section 5.2.1) and then we describe two attacks that disclose the precise location of a target user (Sections 5.2.2 and 5.2.3).

5.2.1 Scenario definition

A *source* person s is using the Service. Another person t (*target*) is using the same service and is located in the vicinity of s in the sense that t is shown to s as a nearby user. User s is the adversary, as she aims at obtaining the position of t with the highest possible precision. To achieve this, s can collude with one or more buddies $c_1 \dots c_n$. In the following, we denote with s, t and c_i both the actors of the scenario and their positions. We also denote with $d(i, j)$ the distance between two users.

To perform the attack, s relies on the knowledge derived by herself and by the colluding buddies from the use of the service. Also, s can use information about her location and the location of colluding buddies c_i as well as data derived from this, like $d(s, c_i)$.

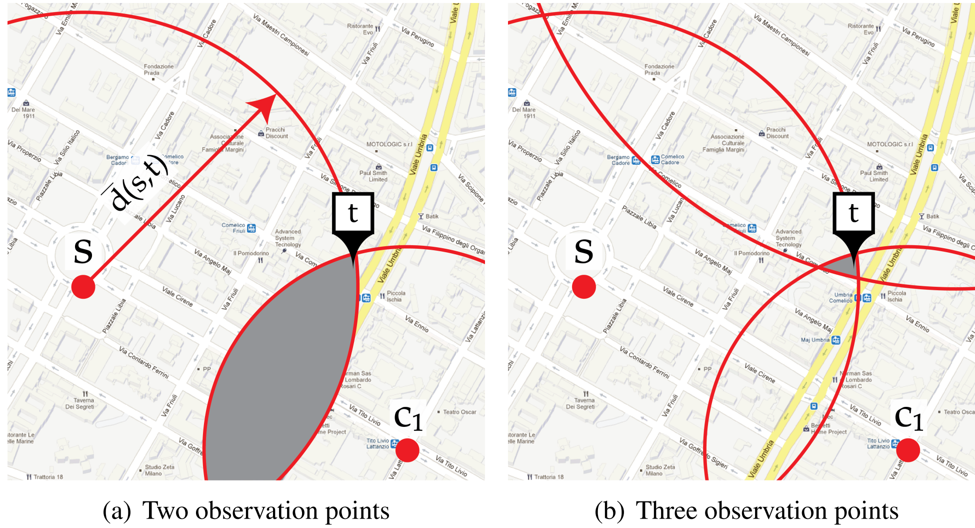
In this work we distinguish two attacks. For some target users, the mobile client of the Service shows the distance of the target from the source user. The distance value is approximated to the upper bound of the distance from t, which we denote with $\overline{d}(s,t)$. In this case, we use a "known distances" attack to retrieve the position of t. In other cases the client does not show the distance of the target from the source user. We call the attack in this case the "unknown distances" attack.

5.2.2 "Known distances" attack

Given the upper bound of the distance $\overline{d}(s,t)$, s can derive that t is located in the circle centered in the position of s with radius $\overline{d}(s,t)$. Clearly, if s also knows $\overline{d}(c_1,t)$ for a colluding buddy c_1, then it is possible to further restrict the position of t to the intersection of the two circles (see Figure 5.1(a)). This is similar to a trilateration attack [105], with the main difference that the exact distance is unknown. If there are more colluding buddies, the position of t can be identified with less approximation (see Figure 5.1(b)).

Although the above attack is straightforward from a theoretical point of view, a number of issues could arise in its practical application: errors due to GPS, approximations in the server-side distance computation, delays in the service provisioning, and so on. To evaluate the feasibility of this attack in practice, we kept

the target user in a fixed position and we used several observations from a moving source user s to simulate a set of colluding buddies. In our experience with the Service, three observations are sufficient to locate t with a good approximation. For example, in Figure 5.1(a) t is located in an area of less than 0.05km^2 while in Figure 5.1(b) the area is about 500m^2.



(a) Two observation points

(b) Three observation points

Figure 5.1: "Manual" execution of the "known distances" attack.

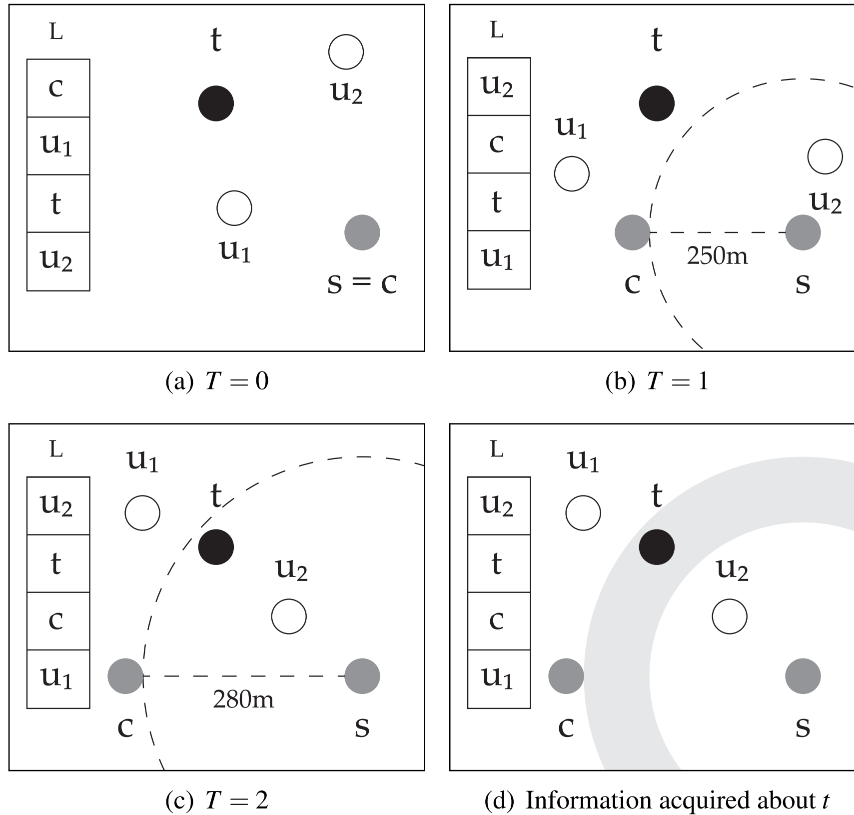
5.2.3 "Unknown distances" attack

When the distance of the target from the source user is unknown, it is not possible to directly compute $\overline{d}(s,t)$. However, we now show how this value can be derived by exploiting the fact that the client shows to the user s the list L of nearby users, ordered according to their distance from s.

In this case, s can discover the approximate distance to t by colluding with another user c as follows: c starts from s moving away from this position, while s periodically monitors L as well as the distance between s and c. As long as c precedes t in the list of users, s knows that c is closer than t. When c happens to be after t in L, then $d(s,t) < d(s,c)$. Since $d(s,c)$ is known, s actually discovers $\overline{d}(s,t)$. Once the approximated distance is discovered, the "known distance" attack can be used to discover the position of t. Note that in this attack we are implicitly assuming that t is not moving during the time of the attack. In Section 5.3 we show that this assumption is not necessary while performing the automated attack.

Example 1 *At time $T = 0$, c is located in the same position as s, hence c is the first element of L (see Figure 5.2(a)). At time $T = 1$, c has moved at a distance*

of 250*m from s, but still precedes t in L (see Figure 5.2(b)). At time $T = 2$, c is shown in L after t and the distance between c and s is* 280*m (see Figure 5.2(c)). The adversary concludes that the distance between s and t is between* 250*m and* 280*m and hence t is located in the gray area of Figure 5.2(d).*



(a) $T = 0$ (b) $T = 1$

(c) $T = 2$ (d) Information acquired about t

Figure 5.2: Example of "unknown distances" attack.

5.3 Attack automation

The attack we illustrate in Section 5.2 is conducted by a human agent by interacting *manually* with the Service. Manual interaction, however, greatly undermine the scalability of the attack. Even assuming that the human user has the ability to feed *false* location information to the Service (i.e., she must not *physically move* to different real-world locations during the attack), manually performing every step needed during the attack may be cumbersome. For this reason, we investigated how to automate the attacks to our target service. To achieve this, we developed

a software agent that automatically communicates with the service provider, pretending to be a mobile client in use by a real user.

5.3.1 Development of ad-hoc client

To develop an ad-hoc client it is first necessary to figure out how a real client communicates with its service provider. To this end, we installed the application on an Android 2.2 system, running inside the Android Emulator [106] and we configured it to connect to the Service with a user that we had previously registered. Then, using the Android SDK functions we "placed" the device in a geographical location and we used the client to update the location and to retrieve nearby users. We repeated this operation several times, each time "placing" the device in a different position. While doing this we captured the network traffic produced by the device and we analyzed it to understand the communication protocol.

By means of CopperDroid and by observing the network traffic we identified a known, non-textual, protocol used to efficiently exchange marhsalled data over a network connection. Since this protocol makes it possible to define *ad-hoc* data types, we create a CopperDroid plug-in to enhance the automatic AIDL unmarshalling to correctly identify different messages. Eventually, we identified most of the messages and we also realized that, for some requests, the service provider does not require authentication, making it possible to obtain important information without registering any user.

After understanding the communication protocol, we developed a Python application capable of communicating with the service provider to compute the following primitive:

$$\mathcal{U} = getNearby(lat, lon, \delta)$$

The primitive takes as input a geographical location $\langle lat, lon \rangle$ and a distance δ, returning a set $\mathcal{U}$ of users reported by the Service as located at most at distance δ from $\langle lat, lon \rangle$.

We observed that the Service does not adopt any security measure, such as encryption, to protect the network traffic generated by its users while using the client. Adopting such a solution, for example by migrating the communication protocol over HTTPS, would undoubtedly increase the overall security of the service, for example preventing an external adversary from sniffing the network traffic. However, note that this is not a limitation of our attack. Indeed, there are many ways we could still retrieve the information we need about the communication protocol. A clever attacker, for example, can reverse engineer the target application to view the source code responsible for the communication protocol. Otherwise, if the application or its user fails to properly validate the SSL certificate, a Man-in-the-Middle [107] attack can be conducted to trick the application into using a fake

certificate, customly created by the attacker that can, consequently, decrypt `HTTPS` traffic. We adopted the latter technique to understand the communication protocol of two friend finder services.

5.3.2 Attack Algorithm

While developing the ad-hoc Python client, we noticed, by means of CopperDroid and its automatic AIDL unmarshalling feature, how data exchanged through the Service's client and server include the *precise* distance between the client's position and nearby users. Figure 5.3 shows an opportunely anonymized selected piece of trace of the application[3].

31	ACCESS PERSONAL INFO	LOCATION	319	aaa.bbb.ccc
32	NETWORK ACCESS	HTTP	319	aaa.bbb.ccc

TS	TYPE	NAME	ADDITIONAL INFO
2012-07-14 05:44:20	SYSCALL	connect	{'host': '::ffff:XXX.XXX.XXX.XXX', 'retval': 0, 'port': 80}
2012-07-14 05:44:21	SYSCALL	sendto	=POST+HTTP%2F1.1%0D%0AHost%3A+XXX.XXX.XXX.XXX%0D%0AConnection%3A+Keep-Alive%0D%0AContent-Length%3A+68%0D%0AContent-Type%3A+application%2Fx-www-form-urlencoded%0D%0Adistance%3D2000%26accuracy%3D0.0...

Figure 5.3: Application Trace

Thus, we customized the *getNearby()* function to return such piece of information too.

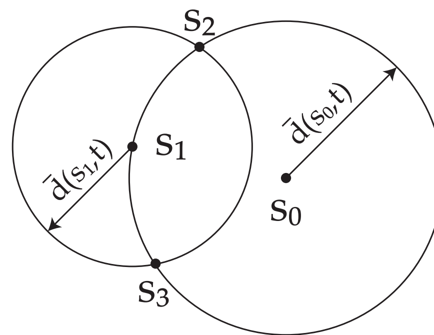


Figure 5.4: Source points chosen by the automated attack.

When precise distances between users are known, the location of the target can be obtained with trilateration. First, we use the *getNearby()* function from a

[3]We do not disclose the unmarshalled data to avoid leakage of application details.

source position s_0 to get the list of nearby users among which we chose the target t. Since the service provides precise distances, we acquire the value $d(s_0,t)$. We then choose a point s_1 on the circle centered in s_0 with radius $d(s_0,t)$ (see Figure 5.4). Again, we use *getNearby()* to retrieve $d(s_1,t)$. Now, let s_2 and s_3 be the two intersections between the two circles centered in s_0 and s_1 with radius $d(s_0,t)$ and $d(s_1,t)$, respectively. We use *getNearby()* for the third time to compute $d(s_2,t)$: if the result is close to zero, then we conclude that t is close to s_2, otherwise t is close to s_3.

This algorithm has the advantage of being simple from a conceptual point of view and to require a constant number of executions of the *getNearby()* primitive; hence, it has a short execution time (a couple of seconds, in our experiments, mainly due to network latency). Also, the position of t can be obtained with high precision. In our experiments, the average error is in the order of a few meters (always less than 10m).

5.4 Privacy Implications

As shown is Section 5.3 it is possible to automate the privacy attacks described in Section 5.2 to discover the position of a target user t under the assumption that t is located close to the source user s. In this section we show how this can be used to achieve three threatening privacy attacks.

5.4.1 “Who is there?” attack

The aim of the “Who is there?” attack is to understand who resides in a given location. Intuitively this attack is particularly threatening when the presence in the chosen location discloses personal data about the user. For example if the adversary chooses a place of worship as target location she can infer, with a certain likelihood, the religious belief of the people at that location. Repeating the observation and checking who is present in that location several times can increase the probability of a correct guess. Technically, the attack can be simply performed by positioning s at the target location and retrieving the users that are closer than a given threshold distance with the *getNearby()* primitive.

5.4.2 “Where is Alice?” attack

Let us consider an adversary that wants to stalk a target user t. Since the approximate position of t is unknown, we cannot directly use the automated attacks presented in Section 5.3 because we do not know where to place s_0. In practice,

we first need to find a position s_0 such that t is “near-by”. Then, we can use the attacks shown in Section 5.3.

To find the position of s_0, we iteratively use the *getNearby()* primitive to “search” for t. In theory, this can be achieved by starting from a given random position and retrieving the nearby users. If t is not in the result, the adversary can chose a new random source position retrieving nearby users to that point. Eventually the location of the target user is found. Clearly, several optimizations are possible. in area where t has already been searched.

In practice, such an attack requires to issue a large number of requests and to retrieve a number of users linear in the total number of users of the service. In our automated attack, we observed that it is possible to retrieve from the service provider about 250 users per second. This means that searching t in a million of users takes about one hour. If the service has hundreds of millions of users this attack is impractical, unless we have some clues about t like profile information (that can be used as filters) or the region where t is likely to be located. For example, considering only female users aged between 20 and 25 years, we have been able to retrieve users in an area of 13km centered in Milan in about half a second.

5.4.3 “Follow Alice” attack

By periodically repeating the “Where is Alice?” attack and storing the results it is possible, after some time, to identify the set of places visited by a target user t. From this “trace”, the attacker can discover t’s home address and workplace and potentially spot t’s real identity.

Clearly the “trace” of places visited by t contains only the locations sent to the service providers by t’s client. Most of the services currently available (including the Service) use clients that send location updates in response to user-initiated actions like log-ins, searches for nearby users or explicit requests. However, some clients allow the user to enable automatic location updates hence periodically sending the user’s location to the service provider, in some cases even when the application is in background. This is a more threatening situation, since a user can be unaware of being disclosing her position.

5.5 Ethical Considerations

Our purpose in this work is to demonstrate *how* an attacker can leverage *publicly available* information provided by a commercial friend finder service in order to *precisely* infer the position of an arbitrary and unaware user. In our attack we do not violate or hack any system. Actually, our objective is not to show

security vulnerabilities in the considered services, but rather to show that it is possible to create an application that pretends to be a client and use it to automate privacy attacks. More specifically, our privacy attack is performed according to the following principles:

1. the attacker does *not* anyhow compromise the servers of the service provider or retrieve otherwise unavailable information;
2. the attacker does *not* interact with her target of choice (e.g., tricking him to visit a specially crafted web page);
3. every information that the attacker uses to infer the position of her target(s) is available either to registered or unregistered users of the service.

This being said, when performing the experiments that are required to proof the soundness of our work, the privacy of real users of the Service must be taken into high consideration. To this end, during our analyses, we targeted only users under our direct control and users of whom we had previously got an explicit authorization.

5.6 Conclusions

In this contribution we have shown that users participating in a "open-buddy" friend finder expose their locations to the public, even if this information is not explicitly given to other users. Indeed, we showed that, after creating an ad-hoc client, it is relatively easy to use public information to spot a user's positions and even to follow a target. While we have developed our ad-hoc client for a desktop computer, it would be possible to run similar code on a mobile device, enabling accessible "on the move" attacks.

The defense against the attack we presented is non trivial. Technically, this is due to the fact that in an "open-buddies" friend finder some location-related information need to be disclosed to the public and the malicious use of this information can easily lead to discover the actual position of the target user. So far, we did not devise any security-related solution to prevent the adversary from learning the communication protocol and actually we have been able to understand and replicate all the protocols of the many service that we investigated. Probably a partial solution to the problem would consist in rendering the attacks more complicated by identifying the attack patterns (e.g., series of requests) and blocking them. However this can hardly be a general solution to the problem.

Similarly, we have not been able to identify any data-management solution to prevent these attacks. One approach that partially enhances users' privacy is

to disclose only approximate position (like the ZIP code, for example). Some services actually implement this solution. While this information is intuitively less sensible than the exact location, disclosing it still causes some privacy issues. Another limit of this solution is that it trades-off privacy for computed distance precision, causing a decrease in the quality of the service.

We have one last consideration about the users' perception of the above problems. We created and published a non-technical video to briefly present the above results to end-users[4]. The video was seen by less than 400 persons. We identified two reasons for this unsatisfying result. First, despite our effort, we had not been able to advertise the video to the correct audience or to make it sufficiently clear. Second, we collected comments showing that apparently people do not have the perception of how much their privacy in endangered by automated attacks. While this is not a strictly technical problem (rather it is a sociological one) we argue that it should be taken into account while devising privacy-preserving solutions.

[4] http://watchyourstep.everywaretechnologies.com/

6 Future directions

In the previous chapters we presented our reasearch work towards to overcome some of the limitations and drawbacks of the current program analysis techniques to perform malware behavior analysis. However, the improvements provided with our novel techniques are not free of limitations. In this chapter we discuss about possible enhancements and extensions aim to augment the capabilities of our solutions in order to provide better and reliable malware analysis results.

6.1 A methodology for testing CPU emulators

EmuFuzzer currently supports IA-32 architecture only. We plan to extend EmuFuzzer to new CPU emulators and architectures. Specifically, pushed by the increasing mobile devices and embedded systems diffusion, ARM architecture is becoming one of the most popular and ubiquitous. Therefore, we would like to improve EmuFuzzer by supporting also ARM architecture. Besides the benefits that could be provided to the emulator by this solution, another main but implicit advantage is related to the security. As all of us know, Android applications run on ARM CPU and are tested on QEMU. QEMU is the environment adopted by many techniques, such as CopperDroid and Google Bouncer [51], to perform dynamic analysis of Android applications. As it happens for x86 architecture, authors of malware leverage discrepancies between the emulated and native environment to evade monitoring. Thus, by means of this extension to EmuFuzzer we could potentially find bugs also on ARM emulators and, thereby, performing the analysis limiting the risk to be circumvented.

6.2 On Reconstructing Android Malware Behaviors

CopperDroid unmarshaller uses reflection, `Parcel`, and `Parcelable` protocols to unmarshal the method parameters of the binder communication. Using CopperDroid and our resource reconstructor (implementing data dependency and forward slicing on low-level traces to group system calls with their associated open commands), we are able to automatically reconstruct high-level Android-specific behavior. In the future we hope to continue to develop CopperDroid's Android malware detection and analysis, possibly by including graph-mining to differentiate between core malicious and benign behaviors [108]. Condensing and summarizing many low-level Android actions into a few high level actions has many advantages with known graph-mining solutions. However, many challenges still exist and addressing them is part of our ongoing research effort. For instance, we can augment the discovery of application behaviors with an hybrid solution which consists of a combination of both static and dynamic analysis. Static analysis could be used to find out possible suspected execution paths that otherwise would be hard to dinamically trigger. Moreover, we would like to enhance our stimulation technique by driving the stimulation based on the application layout without being invasive to instrument the application itself. Finally, we plan to analyze our malware data sets in order to evaluate how the behavior of Android malware is influenced by different OS versions.

7 Conclusion

Malware threaten our daily life and work, spreading from commodity PC to the nowadays ubiquitous smartphone devices that all of us keep in their pocket. Unfortunately, malicious users are always a step ahead of researchers trying to overcome any line of defence and leverage solutions limitations to strengthen their malicious software to last longer undetected.

Though malware analysis and program analysis techniques are part of a longstanding research work, yet they are not ready to face the analysis of malware for mobile devices. Moreover, the analysis environment, which is the base on which a program analysis is performed, lacks of transparency and struggles to faithfully emulate the physical CPU. This limits the precision of the final results. In addition, to deal with the path coverage halting problem, the need of new approaches and heuristics arises. This problem affects not only traditional applications, but even more mobile applications. Due to their nature, user-interaction is mandatory to exercise the application in order to observe the application behaviors.

In this dissertation, we proposed novel techniques to address one of the main problem of the current era that affects our daily life: "*cybercrime*". The research work focused on discovering anomalous behaviors by advanced program analysis techniques and improving the effectiveness of the state-of-art analysis environment.

7.1 A methodology for testing CPU emulators

CPU emulators are complex pieces of software. In this work, we presented a testing methodology for CPU emulators, based on fuzzing. Emulators are tested by generating test case programs and by executing them on the emulated and on the physical CPU. As the physical CPU is assumed to follow perfectly the specifi-

cations, defects in the emulators can be detected by comparing the state of the emulator with that of the physical CPU, after the execution of the test case program. The proposed methodology has been implemented in a prototype, named as EmuFuzzer, and it was used to test five state-of-the-art IA-32 CPU emulators. EmuFuzzer discovered minor and major defects in each of the tested emulators, thus demonstrating the effectiveness of the proposed approach.

7.2 On Reconstructing Android Malware Behaviors

In this work we proposed CopperDroid, a VM-based dynamic system call-centric analysis and stimulation technique to automatically reconstruct the behaviors (OS-specific and Android-specific) of Android malware. We evaluated the performance and effectiveness of our analysis on a large data set of more than 2,900 real world Android malware. Results show how proper external events can actually influence Android malware and lead to the discovery of additional behaviors.

Acknowledgements

Getting a PhD means being thankful to the people that have crossed my street ...how many ...I gotta be formal, at least for the first paragraph. First, I would like to thank my advisor, Prof. DAnilo Bruschi, for his support and suggestions. He has always something good to say. I am also very grateful to my external referees, Prof. Davide Balzarotti, Prof. Giovanni Vigna and Prof. Dawn Song for the time they spent reading this dissertation and their comments that contributed to improve it. In the limbo between formal and informal there's Gigi Sullivan (aka Lorenzo Cavallaro). I could write a long essay about him. He's great. He motivated me a lot. Thanks dude. A big thanks to Lorenzo Martignoni: a mentor and person that change your life. Finally Roberto Paleari a hyper-skilled guy that doesn't need to get introduced and that really helped me a lot during my PhD.

Now we're in the messy side, the informal one. There are so many people I'd like to thank and not enough space. Anyway, folks, here's your turn. I want to thank and remember all those who represent the LaSER crew: Aristide Fattori - my PhD colleague and partner in crime -, Mattia Pagnozzi (Crostatina) - speechless -, Stefano Bianchi Mazzone (il Pelato) - g0t r00t -, Luca Guerra (bruciato) - still reversing? -, Lorenzo Flore (Lollacci) - hangover? -, Mauro Cascella (il Casce) - RSP is not an Italian word -, Srdjan Matic (Smatic) - don't spam -, Fabio Pagani (il Nerd) - pio pio - Andrea Orsini (Winnie) - Dr Jekyll and Mr Hyde -, Salvatore Borgia (salvo) - gentoo addicted and hardware dependent -, Federica De Val (la tipa del tipo) - best web 400 -, Erik Calligari - SSL man but doesn't like to stay in the middle -, and last but not least the LaSER mascot Fabio Pedretti (Joe B.) - you're our PR and party man - ... and again Danilo Bruschi (il DB) - folks, he's the best :) - how many times he made us ROTFL with his jokes?

Finally, in the lovely and loving side, there are the most important persons of my life. First of all, the essential, Chiara. She has always unconditionally believed in me. We began the university together, we met on that bench, and we're still here, *together*. Well... set on that bench there was also my brother... actually he isn't but basically we met the first time when we were 3 years old, Massimiliano Oldani (sgrakkyu). He is always there when you need. To conclude, I wish to thank my family for their help and support.

Bibliography

[1] W. Whitson, "Robberies decrease as cyber crime increases, FBI says." http://www.wmbfnews.com/story/20972727/robberies-decrease-as-cyber-crime-increases-fbi-says, 2013.

[2] P. M. Mell, K. Kent, and J. Nusbaum, "Sp 800-83. guide to malware incident prevention and handling," tech. rep., Gaithersburg, MD, United States, 2005.

[3] L. Page, "Update from the CEO." http://googleblog.blogspot.it/2013/03/update-from-ceo.html, 2013.

[4] McAfee, Inc., "Mcafee threats report: First quarter 2013," tech. rep., McAfee, Inc., 2013.

[5] M. D. Preda, M. Christodorescu, S. Jha, and S. Debray, "A semantics-based approach to malware detection," *SIGPLAN Not.*, vol. 42, pp. 377–388, Jan. 2007.

[6] M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns," in *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, SSYM'03, (Berkeley, CA, USA), pp. 12–12, USENIX Association, 2003.

[7] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *ACSAC*, pp. 421–430, 2007.

[8] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel, "A view on current malware behaviors," in *Proceedings of the 2nd USENIX conference on Large-scale exploits and emergent threats: botnets, spyware, worms, and more*, LEET'09, (Berkeley, CA, USA), pp. 8–8, USENIX Association, 2009.

[9] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, (Washington, DC, USA), pp. 231–245, IEEE Computer Society, 2007.

[10] D. Brumley, C. Hartwig, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin, "Bitscope: Automatically dissecting malicious binaries," tech. rep., In CMU-CS-07-133, 2007.

[11] L. Cavallaro, P. Saxena, and R. Sekar, "On the limits of information flow techniques for malware analysis and containment," in *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, (Berlin, Heidelberg), pp. 143–163, Springer-Verlag, 2008.

[12] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee, "Impeding malware analysis using conditional code obfuscation," in *NDSS*, 2008.

[13] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *Proceedings of the 4th International Conference on Information Systems Security*, ICISS '08, (Berlin, Heidelberg), pp. 1–25, Springer-Verlag, 2008.

[14] U. Bayer, C. Kruegel, and E. Kirda, "TTAnalyze: A Tool for Analyzing Malware," in *15th European Institute for Computer Antivirus Research Annual Conference (EICAR 2006)*, (Hamburg, Germany), 2006.

[15] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual*, Nov. 2008. Instruction Set Reference.

[16] A. Sloss, D. Symes, and C. Wright, *ARM system developer's guide: designing and optimizing system software*. Morgan Kaufmann, 2004.

[17] H. A. Lichstein, "When Should You Emulate?," *Datamation*, vol. 11, pp. 205–210, 1969.

[18] Google Inc., "Android emulator," 2011. http://code.google.com/android/reference/emulator.html.

[19] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, (Chicago, IL, USA), ACM, 2005.

[20] N. Nethercote, *Dynamic Binary Analysis and Instrumentation*. PhD thesis, Computer Laboratory, University of Cambridge, United Kingdom, Nov. 2004.

[21] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, USENIX Association, 2005.

[22] K. P. Lawton, "Bochs: A portable pc emulator for unix/x," *Linux J.*, vol. 1996, Sept. 1996.

[23] I. Preston, R. Newman, and J. Tseng, "JPC: The Pure Java x86 PC Emulator," 2007.

[24] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A Full System Simulation Platform," *Computer*, vol. 35, pp. 50–58, 2002.

[25] "NetBSD/amd64," 2011. http://www.netbsd.org/ports/amd64/.

[26] G. J. Myers, *The Art of Software Testing*. John Wiley & Sons, 1978.

[27] P. Ferrie, "Attacks on Virtual Machine Emulators," tech. rep., Symantec Advanced Threat Research, 2006.

[28] T. Ormandy, "An Empirical Study into the Security Exposure to Host of Hostile Virtualized Environments," in *Proceedings of CanSecWest Applied Security Conference*, 2007.

[29] D. Quist and V. Smith, "Detecting the Presence of Virtual Machines Using the Local Data Table," 2006.

[30] J. Rutkowska, "Red Pill. . . or how to detect VMM using (almost) one CPU instruction," 2004.

[31] T. Raffetseder, C. Kruegel, and E. Kirda, “Detecting System Emulators,” in *Proceedings of Information Security Conference (ISC 2007)*, (Valparaíso, Chile), Springer-Verlag, 2007.

[32] B. P. Miller, L. Fredrikson, and B. So, “An Empirical Study of the Reliability of UNIX Utilities,” *Communications of the ACM*, vol. 33, December 1990.

[33] J. DeMott, “The Evolving Art of Fuzzing,” 2006.

[34] B. Daniel, D. Dig, K. Garcia, and D. Marinov, “Automated testing of refactoring engines,” in *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, (Cavtat near Dubrovnik, Croatia), ACM, Sept. 2007.

[35] R. Kaksonen, “ A Functional Method for Assessing Protocol Implementation Security,” tech. rep., VTT Electronics, 2001.

[36] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

[37] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, “EXE: Automatically Generating Inputs of Death,” in *Proceedings of the 13th ACM conference on Computer and communications security*, (Alexandria, Virginia, USA), ACM, 2006.

[38] P. Godefroid, M. Y. Levin, and D. Molnar, “Automated Whitebox Fuzz Testing,” in *Proceedings of the Network and Distributed System Security Symposium, NDSS*, (San Diego, California, USA), The Internet Society, 2008.

[39] K. Sen, D. Marinov, and G. Agha, “CUTE: a concolic unit testing engine for C,” in *Proceedings of the 10th European software engineering conference*, (Lisbon, Portugal), ACM, 2005.

[40] R. Majumdar and K. Sen, “Hybrid Concolic Testing,” in *Proceedings of the 29th international conference on Software Engineering (ICSE)*, IEEE Computer Society, 2007.

[41] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen, “Directed test suite augmentation: techniques and tradeoffs,” in *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 257–266, 2010.

[42] R. A. Santelices and M. J. Harrold, “Exploiting program dependencies for scalable multiple-path symbolic execution,” in *International Symposium on Software Testing and Analysis (ISSTA 2010)*, (Trento, Italy), pp. 195–206, ACM, July 2010.

[43] R. Paleari, L. Martignoni, G. Fresi Roglia, and D. Bruschi, “N-version disassembly: differential testing of x86 disassemblers,” in *Proceedings of the 2010 International Symposium on Testing and Analysis (ISSTA)*, (Trento, Italy), ACM, 2010.

[44] L. Martignoni, R. Paleari, G. Fresi Roglia, and D. Bruschi, “Testing CPU emulators,” in *Proceedings of the 2009 International Conference on Software Testing and Analysis (ISSTA)*, (Chicago, Illinois, USA), pp. 261–272, ACM, 2009.

[45] L. Martignoni, R. Paleari, G. Fresi Roglia, and D. Bruschi, “Testing system virtual machines,” in *Proceedings of the 2010 International Symposium on Testing and Analysis (ISSTA)*, (Trento, Italy), 2010.

[46] W. M. McKeeman, “Differential Testing for Software,” *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[47] S. K. Lahiri, K. Vaswani, and C. A. R. Hoare, “Differential static analysis: opportunities, applications, and challenges,” in *Proceedings of the Workshop on Future of Software Engineering Research (FoSER 2010)*, (Santa Fe, New Mexico, USA), pp. 201–204, ACM, Nov 2010.

[48] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu, “Differential symbolic execution,” in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, (Atlanta, Georgia, USA), pp. 226–237, ACM, 2008.

[49] K. Taneja, T. Xie, N. Tillmann, J. de Halleux, and P. de Halleux, “eXpress: guided path exploration for efficient regression test generation,” in *Proc. 2011 International Symposium on Software Testing and Analysis (ISSTA 2011)*, (Toronto, ON, Canada), ACM, 2011.

[50] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell, “A Layered Architecture for Detecting Malicious Behaviors,” in *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, Lecture Notes in Computer Science, (Berlin, Heidelberg), Springer, Sept. 2008.

[51] J. Oberheide and C. Miller, “Dissecting the Android Bouncer,” (Brooklyn, USA), SummerCon, 2012. http://jon.oberheide.org/files/summercon12-bouncer.pdf.

[52] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, “Ether: Malware Analysis via Hardware Virtualization Extensions,” in *Proceedings of the 15th ACM conference on Computer and communications security*, (Alexandria, Virginia, USA), ACM, 2008.

[53] T. Garfinkel and M. Rosenblum, “A Virtual Machine Introspection Based Architecture for Intrusion Detection,” in *Proceedings of Network and Distributed Systems Security Symposium, NDSS*, (San Diego, California, USA), The Internet Society, Feb. 2003.

[54] J. Koziol, D. Litchfield, D. Aitel, C. Anley, S. Eren, N. Mehta, and R. Hassell, *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. John Wiley & Sons, 2004.

[55] R. Paleari, L. Martignoni, A. Reina, G. Fresi Roglia, and D. Bruschi, “EmuFuzzer Red-Pills Archive,” 2011. http://security.di.unimi.it/emufuzzer.html.

[56] T. Mai, “Android Reaches 500 Million Activations Worldwide.” http://www.tomshardware.com/news/Google-Android-Activation-half-billion-Sales,17556.html, 2012.

[57] M. Egele, “Invited talk: The state of mobile security,” in *DIMVA*, 2012.

[58] Y. Zhou and X. Jiang, “Dissecting android malware: Characterization and evolution,” in *Proc. of the IEEE Symposium on Security and Privacy*, 2012.

[59] D. Desai, “Malware Analysis Report: Trojan: AndroidOS/Zitmo,” Semptember 2011. http://www.kindsight.net/sites/default/files/android_trojan_zitmo_final_pdf_17585.pdf.

[60] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, “Detecting repackaged smartphone applications in third-party android marketplaces,” in *Proc. of CODASPY*, 2012.

[61] D. Bornstein, “Dalvik VM internals,” in *Google I/O*, 2008.

[62] T. Garfinkel and M. Rosenblum, “A Virtual Machine Introspection Based Architecture for Intrusion Detection,” in *Proc. of NDSS*, 2003.

[63] L.-K. Yan and H. Yin, "DroidScope: Seamlessly Reconstructing OS and Dalvik Semantic Views for Dynamic Android Malware Analysis," in *Proc. of USENIX Security*, 2012.

[64] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. of USENIX ATC*, 2005.

[65] A. Reina, A. Fattori, and L. Cavallaro, "A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors," in *Proceedings of the 6th European Workshop on System Security (EUROSEC)*, (Prague, Czech Republic), April 2013.

[66] X. Zhang, R. Gupta, and Y. Zhang, "Precise dynamic slicing algorithms," in *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, (Washington, DC, USA), pp. 319–329, IEEE Computer Society, 2003.

[67] Y. Zhou and X. Jiang, "Android Malware Genome Project." http://www.malgenomeproject.org/.

[68] Contagio Mobile, "Mila Parkour." http://contagiominidump.blogspot.com.

[69] Android, "Android developer reference." http://developer.android.com/reference/packages.html.

[70] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proc. of CCS*, 2011.

[71] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "Permission evolution in the android ecosystem," in *Proc. of ACSAC*, 2012.

[72] D. Barrera, H. Kayacik, P. van Oorschot, and A. Somayaji, "A methodology for empirical analysis of permission-based security models and its application to android," in *Proc. of CCS*, 2010.

[73] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets," in *Proc. of NDSS*, 2012.

[74] Palmsource Inc., "Open binder documentation." http://www.angryredplanet.com/~hackbod/openbinder/docs/html/index.html.

[75] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," in *Proc. of USENIX OSDI*, 2010.

[76] The Honeynet Project, "Droidbox." https://code.google.com/p/droidbox/.

[77] "Andrubis: A tool for analyzing unknown android applications." http://anubis.iseclab.org/.

[78] Iseclab, "Anubis." http://anubis.iseclab.org.

[79] U. Bayer, C. Kruegel, and E. Kirda, "Ttanalyze: A tool for analyzing malware," in *Proc. of EICAR*, 2006.

[80] R. Xu, H. Saıdi, and R. Anderson, "Aurasium: Practical policy enforcement for android applications," in *Proc. of USENIX Security*, 2012.

[81] H. Lockheimer, "Bouncer." http://googlemobile.blogspot.it/2012/02/android-and-security.html.

[82] J. Oberheide and C. Miller, "Dissecting the Android's Bouncer," *SummerCon*, 2012. http://jon.oberheide.org/files/summercon12-bouncer.pdf.

[83] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "SmartDroid: an automatic system for revealing UI-based trigger conditions in Android applications," in *Proc. of SPSM*, 2012.

[84] S. Anand, M. Naik, H. Yang, and M. Harrold, "Automated concolic testing of smartphone apps," in *Proc. of FSE*, 2012.

[85] B. Gatliff, "Embedding with gnu: the gdb remote serial protocol." http://www.huihoo.org/mirrors/pub/embed/document/debugger/ew_GDB_RSP.pdf, 1999.

[86] C. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using cwsandbox," *Proc. of the IEEE Symposium on Security & Privacy*, 2007.

[87] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "AccessMiner: Using system-centric models for malware protection," in *Proc. of CCS*, 2010.

[88] J. Jenkov, "Java reflection tutorial,"

[89] K. Tam, A. Reina, A. Fattori, and L. Cavallaro, "Automatic reconstruction of android malware behaviors," in *ESORICS*, Springer, 2013.

[90] D. Cogen, "Universal android rooting procedure (rage method)," October 26, 2010.

[91] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin, "Automatically identifying trigger-based behavior in malware," *Botnet Detection*, 2008.

[92] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *Proc. of the IEEE Symposium on Security and Privacy*, 2007.

[93] Android, "Monkeyrunner." http://developer.android.com/tools/help/monkeyrunner_concepts.html.

[94] "Specification language for code behavior," 2008.

[95] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. Mitchell, "A Layered Architecture for Detecting Malicious Behaviors," in *Proceedings of the International Symposium on Recent Advances in Intrusion Detection, RAID, Cambridge, Massachusetts, USA.*, Lecture Notes in Computer Science, Springer, Sept. 2008.

[96] C. Rossow, C. J. Dietrich, C. Grier, C. Kreibich, V. Paxson, N. Pohlmann, H. Bos, and M. van Steen, "Prudent practices for designing malware experiments: Status quo and outlook," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P)*, (San Francisco, California, USA), 2012.

[97] McAfee, "Mcafee." http://www.mcafee.com.

[98] L. Martignoni, R. Paleari, G. Roglia, and D. Bruschi, "Testing CPU emulators," in *Proc. of ISSTA*, 2009.

[99] C. Cadar, D. Dunbar, and D. R. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in Draves and van Renesse [128], pp. 209–224.

[100] S. Mascetti, D. Freni, C. Bettini, X. S. Wang, and S. Jajodia, "Privacy in geo-social networks: proximity notification with untrusted service providers and curious buddies," *The VLDB Journal*, vol. 20, no. 4, 2011.

[101] G. Zhong, I. Goldberg, and U. Hengartner, "Louis, Lester and Pierre: Three protocols for location privacy," in *Privacy Enhancing Technologies*, vol. LNCS 4776, pp. 62–76, Springer, 2007.

[102] L. Šikšnys, J. R. Thomsen, S. Šaltenis, M. L. Yiu, and O. Andersen, "A location privacy aware friend locator," in *Proc. of the 11th Int. Symposium on Spatial and Temporal Databases*, LNCS, Springer, 2009.

[103] L. Šikšnys, J. R. Thomsen, S. Šaltenis, and M. L. Yiu, "Private and flexible proximity detection in mobile social networks," in *Proc. of the 11th Int. Conf. on Mobile Data Management*, IEEE Comp. Soc., 2010.

[104] S. Mascetti, C. Bettini, and D. Freni, "Longitude: Centralized privacy-preserving computation of users' proximity," in *Proc. of 6th VLDB workshop on Secure Data Management*, LNCS, Springer, 2009.

[105] H. L. Groginsky, "Position estimation using only multiple simultaneous range measurements," *Aeronautical and Navigational Electronics, IRE Transactions on*, vol. ANE-6, pp. 178 –187, sept. 1959.

[106] "Android SDK." http://developer.android.com/sdk/index.html.

[107] A. Ornaghi and M. Valleri, "Man in the middle attacks," in *Blackhat Conference Europe*, 2003.

[108] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan, "Synthesizing near-optimal malware specifications from suspicious behaviors," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, (Washington, DC, USA), pp. 45–60, IEEE Computer Society, 2010.

[109] D. Bruschi, L. Cavallaro, and A. Lanzi, "Diversified Process Replicae for Defeating Memory Error Exploits," in *3rd International Workshop on Information Assurance (WIA 2007)*, (San Diego, California, USA), IEEE Computer Society, 2007.

[110] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, , Nguyen-Tuong, and J. Hiser, "N-variant systems: a secretless framework for security through diversity," in *Proceedings of the 15th conference on USENIX Security Symposium*, (Berkeley, CA, USA), USENIX Association, 2006.

[111] J. E. Forrester and B. P. Miller, "An Empirical Study of the Robustness of Windows NT Applications Using Random Testing," in *Proceedings of the* 4th *USENIX Windows Systems Symposium*, Sept. 2000.

[112] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *International Conference on Software Engineering (ICSE 2009)*, (Vancouver, Canada), 2009.

[113] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI'08)*, (Tucson, AZ, USA), June 9–11, 2008.

[114] Intel Corporation, "Intel Software Development Emulator," 2011. http://software.intel.com/en-us/articles/intel-software-development-emulator/.

[115] A. Lanzi, L. Martignoni, M. Monga, and R. Paleari, "A Smart Fuzzer for x86 Executables," in *Proceedings of the 3rd International Workshop on Software Engineering for Secure Systems, SESS, Minneapolis, MN, USA.*, ACM, May 2007.

[116] B. P. Miller, G. Cooksey, and F. Moore, "An empirical study of the robustness of MacOS applications using random testing," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 1, 2007.

[117] B. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl, "Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services," tech. rep., University of Wisconsin-Madison, April 1995.

[118] J. Newsome and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," in *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS ' 05)*, Feb. 2005.

[119] J. S. Robin and C. E. Irvine, "Analysis of the intel pentium's ability to support a secure virtual machine monitor," in *Proceedings of the 9th conference on USENIX Security Symposium (SSYMM'00)*, (Denver, Colorado), USENIX Association, 2000.

[120] V. Sieh and K. Buchacker, "UMLinux - A Versatile SWIFI Tool," in *Proceedings of the 4th European Dependable Computing Conference on Dependable Computing (EDCC-4)*, 2002.

[121] Sun Microsystem, "VirtualBox," 2011. http://www.virtualbox.org.

[122] C. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using CWSandbox," *IEEE Security & Privacy*, vol. 5, no. 2, 2007.

[123] K. Taneja, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Guided path exploration for regression test generation," in *Companion Proceedings of the 31th International Conference on Software Engineering (ICSE 2009), New Ideas and Emerging Results*, pp. 311–314, May 2009.

[124] A. Slowinska and H. Bos, "Pointless tainting?: evaluating the practicality of pointer tainting," in Schröder-Preikschat *et al.* [125], pp. 61–74.

[125] W. Schröder-Preikschat, J. Wilkes, and R. Isaacs, eds., *Proceedings of the 2009 EuroSys Conference, Nuremberg, Germany, April 1-3, 2009*, ACM, 2009.

[126] L. Cavallaro, P. Saxena, and R. Sekar, "On the limits of information flow techniques for malware analysis and containment," in Zamboni [127], pp. 143–163.

[127] D. Zamboni, ed., *Detection of Intrusions and Malware, and Vulnerability Assessment, 5th International Conference, DIMVA 2008, Paris, France, July 10-11, 2008. Proceedings*, vol. 5137 of *Lecture Notes in Computer Science*, Springer, 2008.

[128] R. Draves and R. van Renesse, eds., *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, USENIX Association, 2008.

[129] A. Srivastava, A. Lanzi, J. T. Giffin, and D. Balzarotti, "Operating System Interface Obfuscation and the Revealing of Hidden Operations," in *Proc. of DIMVA*, 2011.

[130] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song, "Juxtapp: A scalable system for detecting code reuse among android applications," in *Proc. of DIMVA*, 2012.

[131] W. Enck, "Defending users against smartphone apps: Techniques and future directions," in *Proc. of ICISS*, 2011.

[132] H. Binsalleeh, T. Ormerod, A. Boukhtouta, P. Sinha, A. Youssef, M. Debbabi, and L. Wang, "On the analysis of the zeus botnet crimeware toolkit," in *Proc. of PST*, 2010.

[133] Android, "Android Documentation." `http://developer.android.com/guide/components/index.html`.

[134] Kaspersky, “Teamwork: How the ZitMo Trojan Bypasses Online Banking Security,” October 2011. http://www.kaspersky.com/about/news/virus/2011/Teamwork_How_the_ZitMo_Trojan_Bypasses_Online_Banking_Security.

[135] Symantec, “Android.Gmaster,” August 2011. http://www.symantec.com/security_response/writeup.jsp?docid=2011-082404-5049-99&tabid=2.

[136] VMware Inc., “VMware.” http://vmware.com/.

[137] X. Jiang and X. Wang, “"Out-of-the-Box" Monitoring of VM-Based High-Interaction Honeypots,” in *Proc. of RAID*, 2007.

[138] J. Rutkowska, “Red pill... or how to detect VMM using (almost) one CPU instruction,” 2004. http://invisiblethings.org/papers/redpill.html.

[139] Google I/O 2012, “Android: More than 400 million devices activated so far âĂŞ daily activation crosses 1 million.”

[140] S. Hofmeyr, S. Forrest, and A. Somayaji, “Intrusion detection using sequences of system calls,” *Journal of Computer Security*, 1998.

[141] PassMark Software, “PassMark Android.” http://www.passmark.com/.

[142] J. Newsome, B. Karp, and D. Song, “Polygraph: Automatically generating signatures for polymorphic worms,” in *Proc. of the IEEE Symposium on Security and Privacy*, 2005.

[143] R. Perdisci, W. Lee, and N. Feamster, “Behavioral clustering of http-based malware and signature generation using malicious network traces,” in *Proc. of the USENIX NSDI*, 2010.

[144] C. U. Center for International Earth Science Information Network (CIESIN) and C. I. de Agricultura Tropical (CIAT), “Gridded population of the world, version 3 (gpwv3),” 2005.

[145] C. Bettini, S. Jajodia, P. Samarati, and X. S. Wang, *Privacy in Location-Based Applications*, vol. 5599 of *Lecture Notes in Computer Science*. Springer, 2009.

[146] G. Ghinita, “Private queries and trajectory anonymization: a dual perspective on location privacy,” *Trans. Data Privacy*, vol. 2, no. 1, 2009.

[147] G. Ghinita, C. R. Vicente, N. Shang, and E. Bertino, "Privacy-preserving matching of spatial datasets with protection against background knowledge," in *Proc. of the 18th SIGSPATIAL Int. Conf. on Advances in Geographic Information Systems*, GIS '10, ACM, 2010.

[148] H. Hu and J. Xu, "Non-exposure location anonymity," in *Proc. of the 25th Int. Conf. on Data Engineering*, IEEE Computer Society, 2009.

[149] K. Liu, C. Giannella, and H. Kargupta, "An attacker's view of distance preserving maps for privacy preserving data mining," in *Proc. of the 10th Eur. Conf. on Principles and Practice of Knowledge Discovery in Databases (PKDD)*, vol. LNCS 4213, Springer, 2006.

[150] P. Ruppel, G. Treu, A. Küpper, and C. Linnhoff-Popien, "Anonymous user tracking for location-based community services," in *Proc. of the Second International Workshop on Location- and Context-Awareness*, vol. LNCS 3987, pp. 116–133, Springer, 2006.

[151] R. Sibson, "SLINK: an optimally efficient algorithm for the single-link cluster method," *The Computer Journal*, vol. 16, no. 1, pp. 30–34, 1973.

[152] "Institut national de la statistique et des études économiques." http://www.insee.fr/.

[153] "Australian bureau of statistics." http://www.abs.gov.au.

[154] M. Scannapieco, I. Figotin, E. Bertino, and A. K. Elmagarmid, "Privacy preserving schema and data matching," in *Proc. of the 2007 ACM SIGMOD Int. Conf. on Management of data*, ACM, 2007.

[155] S. Mascetti, L. Bertolaja, and C. Bettini, "Location privacy attacks based on distance and density information," in *Proc. of the 20th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, ACM, 2012.

[156] W. K. Wong, D. W.-l. Cheung, B. Kao, and N. Mamoulis, "Secure knn computation on encrypted databases," in *Proc. of the 2009 Int. Conf. on Management of data*, SIGMOD '09, ACM, 2009.

[157] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan, "Synthesizing near-optimal malware specifications from suspicious behaviors," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, (Washington, DC, USA), pp. 45–60, IEEE Computer Society, 2010.