5-2013

Transparent and Precise Malware Analysis Using Virtualization: From Theory to Practice

Lok Kwong Yan
Syracuse University

ABSTRACT

Dynamic analysis is an important technique used in malware analysis and is complementary to static analysis. Thus far, virtualization has been widely adopted for building fine-grained dynamic analysis tools and this trend is expected to continue. Unlike User/Kernel space malware analysis platforms that essentially co-exist with malware, virtualization based platforms benefit from *isolation* and *fine-grained instrumentation support*. Isolation makes it more difficult for malware samples to disrupt analysis and fine-grained instrumentation provides analysts with low level details, such as those at the machine instruction level. This in turn supports the development of advanced analysis tools such as dynamic taint analysis and symbolic execution for automatic path exploration.

The major disadvantage of virtualization based malware analysis is the loss of semantic information, also known as the *semantic gap* problem. To put it differently, since analysis takes place at the virtual machine monitor where only the raw system state (e.g., CPU and memory) is visible, higher level constructs such as processes and files must be reconstructed using the low level information. The collection of techniques used to bridge semantic gaps is known as Virtual Machine Introspection.

Virtualization based analysis platforms can be further separated into emulation and hardware virtualization. Emulators have the advantages of *flexibility of analysis tool development* and *efficiency for fine-grained analysis*; however, emulators suffer from the *transparency* problem. That is, malware can employ methods to determine whether it is

executing in an emulated environment versus real hardware and cease operations to disrupt analysis if the machine is emulated. In brief, emulation based dynamic analysis has advantages over User/Kernel space and hardware virtualization based techniques, but it suffers from semantic gap and transparency problems.

These problems have been exacerbated by recent discoveries of anti-emulation malware that detects emulators and Android malware with two semantic gaps, Java and native. Also, it is foreseeable that malware authors will have a similar response to taint analysis. In other words, once taint analysis becomes widely used to understand how malware operates, the authors will create new malware that attacks the imprecisions in taint analysis implementations and induce false-positives and false-negatives in an effort to frustrate analysts.

This dissertation addresses these problems by presenting concepts, methods and techniques that can be used to transparently and precisely analyze both desktop and mobile malware using virtualization. This is achieved in three parts. First, precise heterogeneous record and replay is presented as a means to help emulators benefit from the transparency characteristics of hardware virtualization. This technique is implemented in a tool called V2E that uses KVM for recording and TEMU for replaying and analysis. It was successfully used to analyze real-world anti-emulation malware that evaded analysis using TEMU alone. Second, the design of an emulation based Android malware analysis platform that uses virtual machine introspection to bridge both the Java and native level semantic gaps as well as seamlessly bind the two views together into a single view is presented. The core introspection and instrumentation techniques were implemented in a new analysis platform called DroidScope that is based on the Android emulator. It was

successfully used to analyze two real-world Android malware samples that have cooperating Java and native level components. Taint analysis was also used to study their information ex-filtration behaviors. Third, formal methods for studying the sources of false-positives and false-negatives in dynamic taint analysis designs and for verifying the correctness of manually defined taint propagation rules are presented. These definitions and methods were successfully used to analyze and compare previously published taint analysis platforms in terms of false-positives and false-negatives.

TRANSPARENT AND PRECISE MALWARE ANALYSIS USING VIRTUALIZATION: FROM THEORY TO PRACTICE

by

Lok Kwong Yan

B.S. Polytechnic University, 2004
M.S. Polytechnic University, 2004

Dissertation
Submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in
Computer & Information Science and Engineering

Syracuse University
May 2013

To my grandfather, my parents, my wife, son and daughter.

ACKNOWLEDGMENTS

I would like to express my sincerest gratitute to the people who made this dissertation possible.

To my advisor, Dr. Heng Yin, whose vision, expertise and attention to detail not only guided my research and kept me on track, but also showed me the importance of being a balanced researcher. You taught me that communication is just as important as the research work itself and that I should strive to do my best in both areas.

To my Master's advisor and friend, Dr. Nasir Memon. You introduced me to computer security and planted the seed that has grown into this dissertation. My career would have been completely different without your initial and continued support and encouragement.

To my labmates, Aravind, Mu, Manju, Eknath, Qian, Andrew and Xunchao. Not to undermine your help in refining my ideas, conducting research and gathering data, but what I appreciate most are the social interactions with you. Our more mundane, non-technical conversations helped relieve the many stresses of working in an academic research lab.

To my previous supervisor at the Air Force Research Laboratory, Information Directorate, Mr. Duane Gilmour, who understood the importance of dedicating time towards research and was instrumental in ensuring that I had the time to do so.

Most important of all, to my wife Victoria. This dissertation would not have been possible without your behind-the-scenes support. Taking care of our infant son while

balancing all of your other responsibilities must have been more difficult and stressful than my research. For that, I am forever indebted to you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

It is estimated that malware (malicious software) costs United States consumers $2.3 billion and led to the replacement of 1.3 million personal computers (PCs) in 2010 [1]. Cybercrime, fueled by malware, was also estimated to have had a world-wide financial impact of $118 billion [2] in the same year. As adoption of mobile platforms increased, so has the prevalence of mobile malware. According to McAfee, approximately 450 mobile malware samples, 400 of which target the Android platform, were identified in the 4th quarter of 2011 [3]. This is a 3.5x increase from the 3rd quarter and is expected to grow in 2012 [4]. The increase in the number of malware samples is also accompanied with new forms of fraud. lookout, a mobile security firm, has pointed out that "the most prevalent Toll Fraud malware family [FakeInst] has netted an approximate $10 million for its makers" from September 2011 to June 2012 [5].

To mitigate the impacts of malware, anti-virus companies and researchers have used static and dynamic analysis tools to understand malware behavior so that countermeasures can be developed. In a survey of dynamic malware analysis techniques and tools, Egele et al. described the differences between static and dynamic analysis tools and outlined five dynamic analysis techniques (Function Call Monitoring, Function Parameter Analysis, Information Flow Tracking, Instruction Trace and Autostart Extensibility Points) that have been implemented using three different strategies (User/Kernel Space, Emulator and Virtual Machine) [6]. Of the techniques, taint analysis and instruction tracing provide the

most detailed information; however, there are limitations. The effectiveness of taint analysis is limited by the *precision* of taint propagation rules and while it is trivial to generate an instruction trace in an emulator, emulators lack *transparency* (and thus can be detected and evaded), and introduce *semantic gaps*.

This dissertation addresses these limitations and shows a fine-grained dynamic malware analysis platform that is transparent and supports precise taint analysis is feasible. The thesis is transparent malware analysis platforms with precise taint tracking rules can be realized using virtualization technologies. Three arguments are used to support this hypothesis: hardware virtualization can be used to make emulation transparent, semantic gaps can be bridged and the precision of taint propagation rules can be formally analyzed.

Transparency Emulators are designed to imitate the behavior of another system. For example, the CPU emulator in QEMU [7, 8] is capable of emulating x86, ARM and other CPU architectures on top of different host architectures (e.g., x86 on x86 and x86 on ARM). If the CPU emulator is perfect, then the imitated behavior exactly mirrors that of a real hardware CPU and the emulator is considered *transparent*. However, transparency is difficult to achieve in practice due to emulation bugs, different CPU models and errata [9–12].

This dissertation proposes *precise heterogeneous record and replay* (PHRR) as a new technique to make emulation based dynamic analysis more transparent. The key idea is to separate a malware's execution from the analysis task. The malware is allowed to execute in a hardware virtualized environment where the above mentioned transparency issues due to emulation do not exist since execution takes place on a real CPU. The execution is

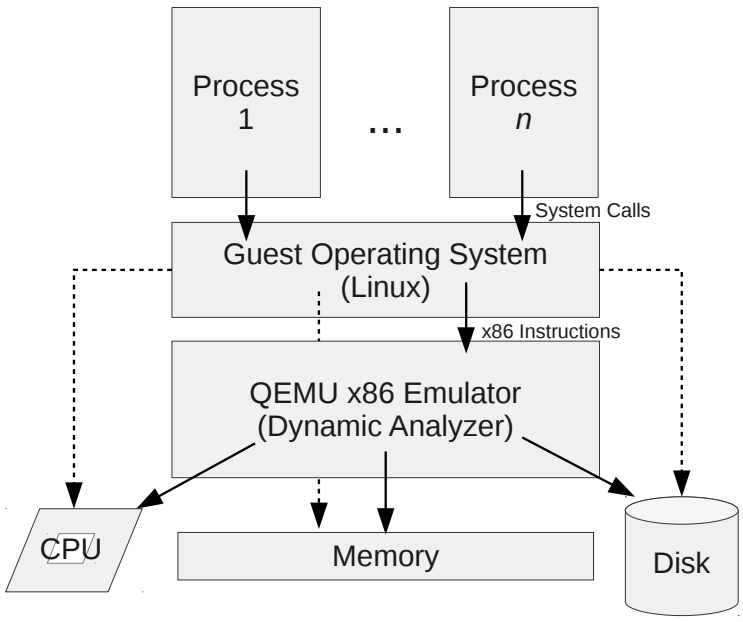

Fig. 1.1.: A high level architectural diagram of an emulation based dynamic malware analysis platform.

recorded such that all inputs and any potentially deviant behavior between the emulator and the real CPU are logged. In this manner, the emulator can better imitate the real CPU during analysis. Precise heterogeneous record and replay is implemented in a tool called V2E that uses the Kernel Based Virtual Machine (KVM) [13] as the recorder and TEMU [14–16] as the replayer. The details are presented in Chapter 3.

Semantic Gaps The semantic gap problem arises when analysis takes place at a lower abstraction layer than the analysis target resulting in a loss of higher level details. Take the emulation based dynamic analysis platform depicted in Figure 1.1 as an example. The solid arrows in the figure represent the interfaces between the different abstraction layers. Processes use *system calls* to interact with the operating system and the operating system uses x86 instructions to access hardware resources. These x86 instructions are emulated,

meaning the emulator mediates all accesses to hardware resources by the guest operating system and executes the guest instructions on its behalf. If emulation was not used, then the operating system will have direct access to the resources (dashed arrows).

It is clear from the figure that system calls are the interfaces between processes and the operating system, and not between the operating system and the emulator. Hence, an analyzer situated at the emulator that seeks to log all system calls made by "Process 1" will have difficulties. Since the emulation layer can only access the raw CPU and memory states, the concept of a process (an operating system construct) is limited to the value of the CR3 register, which contains the physical address of the top level page table. Also,the concept of Linux system calls is limited to the `int 0x80` instruction that is used to invoke system calls. Additionally, while the CR3 values can be used to distinguish between different processes, specific information such as which CR3 value represents "Process 1" is not available at the emulation layer. To retrieve this information, and thus bridge the semantic gap, researchers have proposed Virtual Machine Introspection (VMI) techniques that can identify and interpret specific data structures from raw memory [17–19]. While these techniques have proven useful for bridging the semantic gap between the emulator and the guest operating system, Android, a mobile platform, has multiple gaps that must be bridged.

Unlike traditional desktop systems where applications run directly on top of the operating system, Android applications run in a Java Virtual Machine - known as the Dalvik Virtual Machine - and use JNI (Java Native Interface) to interface with native libraries. Consequently, Android malware can contain Java and native components that cooperate to achieve a common goal. Therefore, there are two levels of semantic

information that must be reconstructed in order to adequately analyze Android malware. First is the native level that includes the Linux kernel and second is the Java level that is interpreted using the DVM.

Bridging these two gaps involves understanding the different kernel data structures used to store pertinent information (e.g., processes) and the data structures and logic of the DVM (e.g., Java objects). These details along with the design of a new emulation analysis platform that seamlessly binds the two views together so that a sample's Java execution, native execution and interactions between them can be studied are presented in Chapter 4. These ideas are implemented in a new analysis tool name DroidScope, which is also discussed in the same chapter.

Precise Taint Analysis Dynamic taint analysis is designed to track information flows through data items of interest and is a key binary analysis technique [6]. In taint analysis, data items are labeled as "tainted" or "untainted" and this taint (label) is propagated from data item to data item as the program being analyzed is executed. The rules that determine when and how taint is to propagate are defined in a *taint propagation policy*. Taint analysis is supported by both V2E, through TEMU, and DroidScope.

The propagation policies have been widely researched with different design patterns that vary taint-granularity (e.g., byte versus bit tainting [both TEMU and DroidScope labels taint per byte of data]), analysis-granularity, (e.g., propagating through ARM instructions like DroidScope or an Intermediate Representation [IR] like TEMU), and special case support (e.g., TEMU has a special rule to propagate taint through the bit shift operations without introducing false-positives while DroidScope does not). The

relationships between these design parameters and the accuracy (a measure of false-negatives) and precision (a measure of false-positives) of taint analysis are analyzed in Chapter 5 using formal methods.

In particular, a formal model of dynamic taint analysis is defined based on the concept of information flow or noninterference [20]. The model is then used to prove that propagating taint at the byte-level can introduce false-positives when compared to propagating taint at the bit-level and similarly, propagating taint through an IR can introduce false-positives when compared to propagating taint through the native instruction the IR is used to emulate. Methods for automatically generating a default policy without false-negatives and for determining the accuracy and precision of manually defined taint propagation rule are also presented.

Summary In short, this dissertation presents concepts, techniques, methods and proofs that can be used to design and build transparent and precise malware analysis platforms. DroidScope illustrates that emulation based fine-grained binary analysis can be performed on both mobile and desktop malware by bridging the two semantic gaps in Android, V2E shows that emulators can be made more transparency using hardware virtualization, and Chapter 5 provides insights into the fundamental sources of imprecision in dynamic taint analysis designs and methods for building and verifying precise taint propagation rules and taint analysis platforms.

The rest of the dissertation is organized as follows. Background information on malware analysis, virtualization, and taint analysis is presented in Chapter 2. V2E and DroidScope are presented in Chapters 3 and 4 respectively. The formal model for understanding the

relationship between dynamic taint analysis implementations and precision, and its application towards analyzing the precision of previously published taint analysis implementations are presented in Chapter 5. Chapter 6 summarizes the results. Future work is discussed throughout the dissertation.

2. BACKGROUND

2.1 Virtualization

Virtualization is a computing concept where a device, component, or system is simulated for use by another device, component or system. To clarify the concept, Smith and Nair [21] state that "a discussion of [Virtual Machines] is also a discussion about computer architecture in the pure sense of the term." Then "architecture, as applied to computer systems, refers to a formal specification of an interface in the system" that can be implemented using multiple abstraction layers, each with its own interface (i.e., sub-architecture).

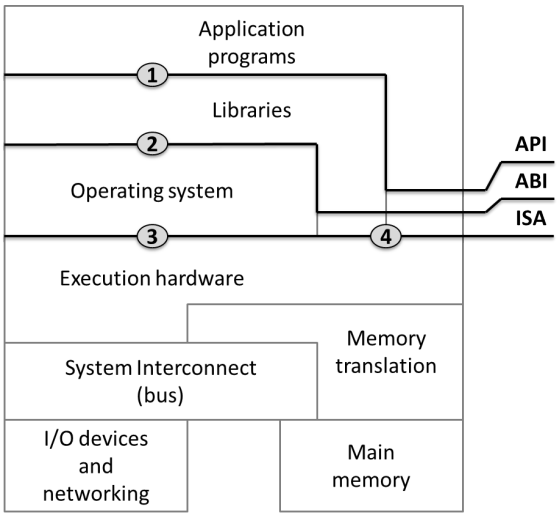

Fig. 2.1.: Computer System Architecture [21]

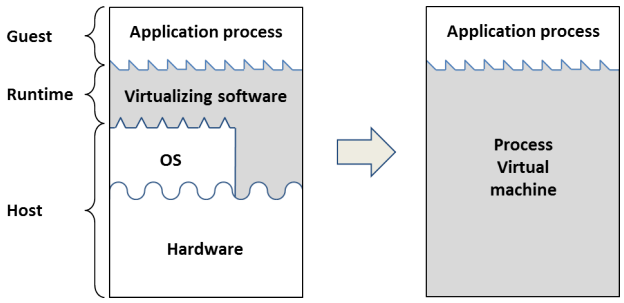

Fig. 2.2.: Process VM [21]

For example, a notional computer system architecture is depicted in Figure 2.1 [21]. While the computer system architecture consists of many different interfaces, four are notable. The Instruction Set Architecture (ISA) divides the hardware and software layers. It is further separated into the privileged system ISA (3) and unprivileged user ISA (4). The Application Binary Interface (ABI) is used by user programs to access system resources directly using the user ISA and indirectly using the system call interface (2). Finally, the Application Programming Interface (API) (1) abstracts away some details of the ABI through the use of libraries.

Given the multiple abstraction layers, a virtual machine for a particular abstraction layer is defined as the simulation of the interface beneath it. This definition is shown in Figure 2.2 [21] where a process virtual machine uses the "virtualization software" to simulate the ABI or API layers that the process uses. The figure also shows the virtualization software separating the *host* from the *guest*. More specifically, the virtualization software runs on top of the *host* and the *guest* runs on top of the virtualization software.

Special consideration is given to system virtual machines (Figure 2.3) that virtualize the ISA because they have been heavily researched since the 1960's and are used in this

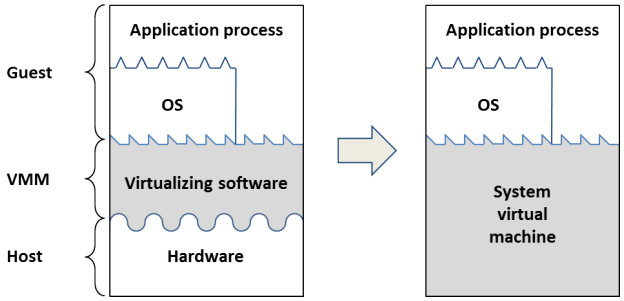

Fig. 2.3.: System VM [21]

dissertation. In this case, the virtualization software is known as a the *Virtual Machine Monitor* or VMM; it is also known as a *hypervisor*. There are two research directions, *emulators* and *virtual machines*. Goldberg [22] defined emulators as efficient "simulators for dissimilar machines" and virtual machines as "efficient simulators for multiple copies of a machine on itself". Therefore, what distinguishes an emulator from a virtual machine is the fact that virtual machines simulate themselves, while emulators simulate other machines.

This distinction between emulators and virtual machines is important; however, the definitions have not been universally applied. For example, the Java Virtual Machine [23] and Dalvik Virtual Machine [24] are emulators according to the definitions, but are known as virtual machines. Since emulators can be considered as a special type of virtual machine [25], virtual machine will henceforth be used to describe the generic concept. *Hardware virtualization* and *hardware virtual machines* will be used to describe virtual machines that simulate themselves. In short, emulators and hardware virtual machines are two different types of virtual machines.

2.1.1 Hardware Virtualization

There are three properties of hardware virtualization: *efficiency*, *resource control* and *equivalence* [26]. *Efficiency* requires the guest-issued instructions in the user ISA (interface (4) in Figure 2.1) to be executed directly on hardware and *resource control* dictates that the guest-issued instructions in the system ISA (3) must be mediated by the VMM. *Equivalence* requires that any program should have the same execution behavior in the virtual machine as it would on real hardware[1]; this is also known as *transparency*. Machines with these properties have been developed in the past [22], but it was only until recently that hardware virtualization for the x86 and ARM architectures became available.

The x86 architecture uses a ring-based access control model to separate the user ISA from the system ISA. Since the system ISA is associated with the most privileged ring (Ring 0), the resource control property requires the VMM to either run at Ring 0 while the guest executes at a lower protection level or the VMM to reside in a more privileged ring [27]. The first solution is not ideal since privileged guest instructions must be trapped and emulated [28], which reduces performance. The second solution is realized by introducing a new guest execution mode (*non-root mode* in Intel platforms [29] and *guest mode* in AMD platforms [30]) with its own distinct set of protection rings and CPU states [27]. A similar technology has been introduced for the ARM architecture [31]. Intel nomenclature will be used in the rest of this dissertation for consistency; however, the concepts apply to AMD and ARM as well.

[1]Timing and resource availability problems are assumed not to exist

The relationship between root mdoe and non-root mode is similar to the one between the Operating System (OS) and a process. In order to execute a guest, the VMM (executing in *root* mode) first instantiates a Virtual Machine Control Structure (VMCS) (one per guest) that is used to maintain the state of the guest CPU (i.e., the non-root mode state) as well as control guest access to resources. Once instantiated and loaded, the VMM can use the `vmlaunch` instruction to enter non-root mode for the first time.

The guest continues to execute in non-root mode until an access violation (as configured in the VMCS) occurs which leads to a *VMExit*. As part of the VMExit event, the non-root mode CPU state and *exit reason(s)* are saved into the VMCS and the CPU is transitioned back into root mode. The VMM can then read the exit reason using the new `vmread` instruction, address the violation, update the guest CPU state (in the VMCS) using the new `vmwrite` instruction as necessary and continue executing the guest using the `vmresume` instruction. `vmresume` will load the guest CPU state from the VMCS and transition back into non-root mode. More detailed information can be found in Intel's Software Developer's Manual, Volume 3C [32].

Two-Dimensional Paging Virtual memory is a virtualization technique that exposes a consistent representation of memory to processes [21] and even to guests in virtual machines [25]. Paging and segmentation are two virtual memory implementations that are supported by the x86 architecture in a hardware Memory Management Unit (MMU). Segmentation is controlled through special registers, which have been virtualized as part of the Intel VT [27, 29] and AMDV [30] hardware virtualization extensions.

Paging is implemented using memory-resident page tables that map virtual addresses to physical addresses. Since physical memory is shared between the guest and the host, access to the page tables by the guest must be mediated by the VMM (the resource control property), which can degrade performance. The concept of Two-Dimensional Paging (TDP) was introduced to reduce the number of transitions from non-root mode to root mode due to guest page faults [33].

Two page tables are used in TDP, one for the host and one for the guest. A first level page table is used by the guest to maintain mappings between its virtual addresses (*guest virtual addresses* or GVA) to its physical addresses (*guest physical addresses* or GPA). The guest has full control over this page table and handles the corresponding page faults. A second level page table is used to translate the guest physical addresses to *host physical addresses* (HPA). Page faults at this level (TDP faults) are handled by the VMM in root mode. In a way, the guest physical addresses are *host virtual addresses* (HVA).

During execution in non-root mode, a GVA will first be translated into a GPA using the first level page table. If a page fault occurs, then the guest OS's page fault handler will be invoked to handle the fault. If the translation is successful, then the GPA will be translated into a HPA using the second level page table. If a TDP fault occurs, then the CPU will automatically transition from non-root mode into root mode where the VMM can handle the fault. Once handled, the VMM will transition the CPU back into non-root mode to continue executing the guest.

TDP has been implemented as Extended Page Tables in Intel [29], Nested Page Tables in AMD [34] and System MMU in ARM v7-A architectures [31].

2.1.2 The QEMU Emulator

QEMU [7, 8] is a whole-system emulator that includes a number of subsystems: CPU emulator, emulated devices, generic devices, machine descriptions, debugger and user interface. Its many subsystems have been used as the basis for a number of commercial virtualization software products including KVM, Xen [35, 36], VirtualBox [37], and the Android emulator [38]. Hence, QEMU is considered to be mature; emulation bugs are rare. The full documentation can be found on the project's webpage at `http://www.qemu.org`. Further discussions on QEMU will be focused on the CPU emulator subsystem. For brevity, QEMU will refer to the CPU emulator unless otherwise noted.

QEMU uses Dynamic Binary Translation (DBT) to emulate different instruction set architectures (e.g., x86, ARM and PowerPC). In brief, software emulation based on DBT works as follows. When the emulator is about to execute a block of guest code for the first time, it translates that code block into a piece of translated code in the host's ISA. The translated block is also stored into a code cache to improve performance. When the same code block needs to be emulated in the future, the emulator skips the translation procedure, directly fetches the translated code from the code cache and executes it. Older versions of QEMU (version 0.9.1 and older) used a target-specific translator called DynGen [7] that directly translates guest instructions into host instructions. Newer versions use TCG (Tiny Code Generator), which first translates guest instructions into an intermediate representation (TCG-IR), then compiles the TCG-IR blocks into host instructions for execution.

In either case, special care is needed to emulate memory accesses. The *softmmu* is a software implementation of the Memory Management Unit, which uses the guest's page tables to translate guest virtual addresses to guest physical addresses. To speed up the address translation, the *software Translation Look-aside Buffer (TLB)* is implemented as a cache for the address translation results. Since QEMU is a user-space program, the guest's physical memory is mapped into the QEMU process' virtual memory space. Thus, as an optimization, QEMU's software TLB caches GVA to HVA (in the QEMU process' context) translations instead of GVA to GPA translations.

QEMU is designed to be fast. Consequently it deviates from a real CPU in at least the following ways. First, a block-by-block translation procedure is introduced; this is in contrast to the instruction-by-instruction procedure used in real hardware. This translation procedure is normally invisible to the emulated execution, except when the block being translated crosses the page boundary and the following page is not present in the page table. This naturally leads to a page fault; however, there is a timing difference. The page fault occurs at the first instruction of the block in QEMU instead of at the instruction that crosses the page boundary - the expected behavior - in real hardware.

Second, for efficiency, QEMU performs a lazy calculation of flags: a flag is calculated only when it is needed. Take the following x86 instructions for example: "`cmpl $1, %eax; jz 0x401020;`". On real hardware, the *zero flag* will be set or cleared after the `cmpl` instruction is executed. That is, if the `eax` register is `1` then the ZF bit in the EFLAGS register will be set. If `eax` is not `1` then ZF will be cleared. In QEMU, EFLAGS is not calculated when the first instruction executes. Additionally, on the second instruction, only ZF is calculated to determine which branch to take, the other flags are not calculated until

they are needed or at the end of a block. This lazy approach is good for efficiency but can be exploited to detect emulation.

Third, again for efficiency, interrupts are checked and served only at the block boundary. In contrast, on real hardware, interrupts may happen at any time. As a result, timing differences due to interrupt handling can be observed using a sequence like "`rdtsc; mov %eax, %ebx; rdtsc;`." Since the `rdtsc` instruction stores the current time stamp counter (a count of the number of clock cycles since the CPU has been reset) into the `eax` register, the sequence effectively stores the before and after counts into the `ebx` and `eax` registers respectively. By looping the sequence until the number of CPU cycles is greater than a relatively large threshold (e.g., the number of cycles to execute an interrupt handler), the presence of an interrupt in between the instructions can be detected. An emulator that only issues interrupts at the block boundary will remain in the loop endlessly, whereas a real CPU will eventually exit when an interrupt occurs within the sequence.

In addition to the above discrepancies that are unique to dynamic binary translation, several more are common due to emulation difficulties. First of all, some special-purpose instructions (e.g., System Management Mode and Trusted Execution Technology instructions) are hard to emulate and thus have not been implemented in software. As an example, QEMU 0.9.1 did not simulate the privileged "Resume from System Management Mode" instruction `rsm`. As a result, a malware sample that execute an `rsm` instruction will receive an illegal instruction exception, which is incorrect.

Accurate CPU timestamp emulation is difficult as well. For simplicity, QEMU fetches the timestamps from the host. On the one hand it will address the interrupt detection example discussed above, but on the other hand it introduces a different detection

technique. Since guest instructions are translated into one or more host instructions, emulation consumes more CPU cycles than real hardware. The slowdown can also be used for emulation detection.

Finally, the logic of checking for and raising exceptions in the hardware is fairly complex and thus the software emulation of this logic is often error-prone [9].

2.1.3 The Kernel Based Virtual Machine (KVM)

KVM is a virtual machine monitor that has been integrated into the Linux Kernel since version 2.6.20 and supports a number of hardware virtualized architectures including Intel VT, AMDV and ARM virtualization. Unlike other VMMs (such as VirtualBox and Xen), KVM is more of an accelerator than a full fledge virtualization solution. It abstracts away the details of the different virtualization extensions and exports a uniform interface through the "/dev/kvm" device. It is up to a user-space application to manage the guest resources. In this way, user-space virtualization solutions such as QEMU can use KVM to attain efficiency, resource control and equivalence. The notional architecture is depicted in Figure 2.4.

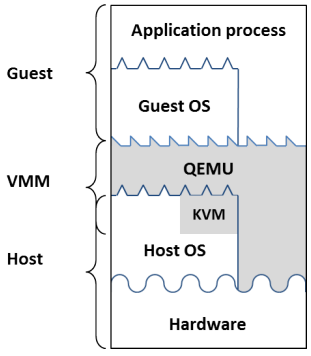

Fig. 2.4.: System Virtual Machine using KVM and QEMU

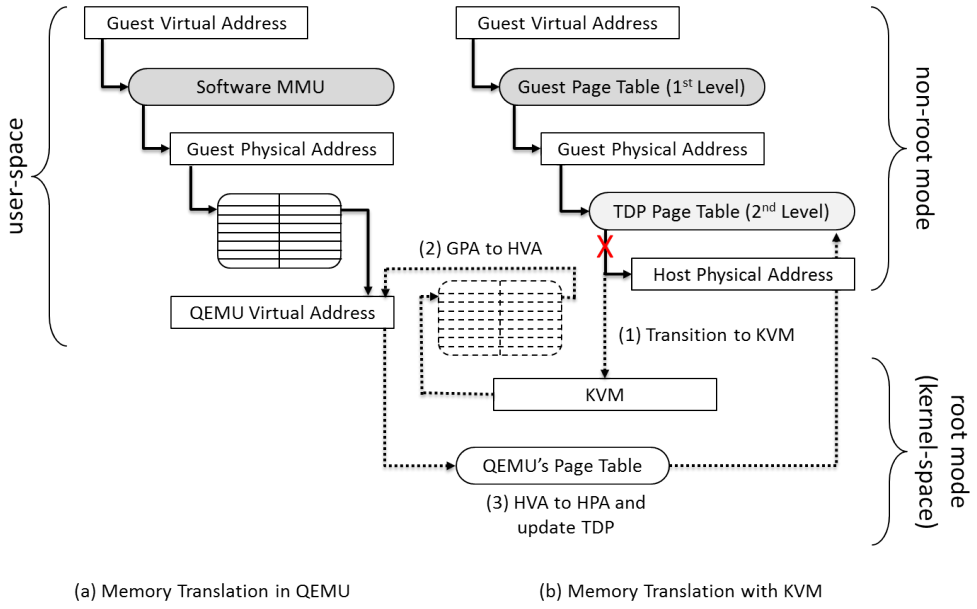

Fig. 2.5.: Memory translation in QEMU without KVM (a) and with KVM and TDP (b)

When KVM is paired with QEMU (the system emulator), the CPU emulator, software MMU and soft TLB are effectively disabled. The CPU emulator is no longer needed because guest instructions execute natively on real hardware in non-root mode and the software MMU and TLB are replaced with TDP. Using TDP with QEMU introduces a complication though.

Since the guest's physical memory space is mapped into the QEMU process' virtual memory space (i.e., GPA to HVA), it is incompatible with the EPT tables that translates GPA to HPA. The multi-stage process depicted in Figure 2.5 is used to address this issue. First, the QEMU process reports the mappings (the table with solid lines) between GPAs and HVAs to KVM (the table with dotted lines). Then, when a TDP fault occurs, KVM takes the TDP faulting address (a GPA) and translates it into the corresponding HVA (2). The HVA is subsequently translated into a HPA using the QEMU process' page table and the new GPA to HPA entry is added to the EPT table (3).

MMU-Notifiers Since QEMU runs as a user level process, its memory can be remapped and swapped to disk by the kernel. Consequently, this leads to inconsistencies in the EPT tables since they continue to the old physical page frame until updated. KVM uses MMU-Notifiers to prevent these kinds of problems. MMU-notifiers are callback functions that kernel modules (such as KVM) can register with the Linux MMU subsystem. In this way, the registered modules are notified of any changes to the page tables before the changes are committed. In the case of KVM, the MMU-notifier callback functions ensure that the EPT tables are consistent with QEMU's page tables.

2.2 Dynamic Malware Analysis

Malware analysis is an important step towards defending against malware. Given a piece of unknown malware, the objective of malware analysis is to reverse engineer it and quickly reveal its inner workings so that countermeasures can be implemented. Malware analysis techniques can be separated into two categories depending on whether the sample is executed during analysis. It is not executed in static analysis and is executed in dynamic analysis. Egele et al. [6] outlined the differences between static and dynamic analysis and provided motivations for the latter.

In general, static analysis has the advantage of code coverage, but obfuscation techniques can be used to hide the code and thus make static analysis less productive. On the other hand, dynamic analysis has the advantage of being unaffected by obfuscation since the code must be de-obfuscated during execution. However, it has the disadvantage of only being able to analyze the execution path taken - that is, it lacks code coverage.

Researchers have presented techniques for exploring multiple execution paths at runtime, but they are not perfect solutions [39–41]. In the end, static and dynamic analysis are complementary techniques that are used together. This dissertation focuses on dynamic analysis.

There are three different implementation strategies, User/Kernel Space, Emulator and Hardware Virtualization[2], each with their advantages and disadvantages [6]. DynamoRIO [42], Pin [43, 44], Valgrind [45] and SR-Dyninst [46] are powerful dynamic instrumentation tools for analyzing user-level programs. They cannot be used to analyze kernel malware. Cobra [47] is a malware analysis platform implemented in a Windows kernel module. It uses a technique called localized execution to instrument and inspect malware behavior. The localized execution technique is, in spirit, similar to dynamic binary translation.

The advantage of virtualization based analysis over User/Kernel Space implementations is *isolation*. Since the analysis tools are implemented at the VMM and the samples execute in the less privileged guest, virtualization isolates the tools from the samples. This allows the tools to analyze privileged malware (ones that execute in the guest kernel) while at the same time making it difficult for the malware to disrupt analysis [6].

The main disadvantage is the loss of semantic contextual information since the analysis component is moved out of the box. Virtual Machine Introspection [17–19] has been used to bridge the semantic gap. Many virtualization based analysis platforms have been implemented using emulation [41, 48–59] and hardware virtualization [60, 61]. These platforms provide the basic functionality to *introspect* (i.e., read and interpret the guest

[2] "Virtual Machine" is replaced with "Hardware Virtualization" as discussed in Section 2.1

state) and *instrument* (i.e., intercept and write the guest state) the malware samples executing within the virtual machine. Analysis *plugins* that implement different dynamic analysis techniques (e.g., Function Call Monitoring, Function Parameter Analysis, Information Flow Tracking, Instruction Trace and Autostart Extensibility Points) are loaded onto the platforms to perform the desired analysis function.

Between the two virtualization based implementation strategies, emulation has the additional benefits of *flexibility* for analysis plugin development and *efficiency* for fine-grained analysis. In hardware virtualization, the VMM is required to be a privileged program so it can manage privileged system resources such as the TDP page table. Emulators such as QEMU can be implemented in user-space. Thus, the analysis plugins built on top of emulators can benefit from user-space libraries making them more flexible. As an example, the Linux kernel does not have an interface to read and write files from kernel [62, 63]. In V2E, the log is written to disk by a user-space program through the use of a shared memory buffer between the VMM's (kernel's) memory space and the program's memory space.

Emulators are also more efficient for fine-grained analysis such as instruction tracing [6]. Tracing an instruction using hardware virtualization requires a transition from non-root mode to root mode and then back for every single instruction - a technique known as single stepping - which can degrade performance. A test of single stepping in KVM showed an approximately 3000 times slowdown in performance.

There is a major disadvantage of emulation though. Emulators lack transparency.

2.2.1 Transparency

Since it is extremely difficult (if not completely infeasible) to emulate every aspect of real hardware, malware can take advantage of these discrepancies to detect the emulated environment and stay dormant to avoid analysis. Some of the problems were raised in the QEMU section, but the general problem of emulation detection and mitigation techniques has been investigated extensively [9–12].

EmuFuzzer [10] and PokeEmu [9] are two noteworthy projects that sought to discover emulation bugs. EmuFuzzer discovered bugs in a randomized fashion using fuzz testing and PokeEmu is a follow-on work that discovers them systematically. PokeEmu uses symbolic execution [64] to explore all instruction emulation paths in two different emulators and generate a set of input-output behaviors that covers 95% of x86 instructions emulated. The set was then used to automatically generate a collection of programs that, given the same input, produces different output when executed on different emulators and real hardware. Over sixty thousand such programs were generated to identify differences between QEMU and real hardware. The differences were found in registers, memory, floating point registers and exception behavior; however, it is unknown whether all of the differences can be corrected in QEMU. It follows that the transparency problem remains unless all of these differences can be patched.

To tackle the problem of transparency from a different perspective, Dinaburg et al. [60] formally defined transparency/equivalence, proved that, except for timing differences, hardware virtualization achieves perfect transparency and implemented a new analysis

platform called Ether. However, Ether, being a hardware virtualization based implementation, is not as flexible or efficient as emulation based implementations.

In order to successfully emulate anti-emulation malware, Balzarotti et al. [65] used an automatic method to detect the anti-emulation behavior by comparing how a piece of malware behaves on real hardware and how it behaves in an emulated environment. Similarly, Kang et al. [66] used a differential analysis method that compares two execution traces, one from Ether and the other one from QEMU. By performing trace alignment, this technique is able to automatically detect the root cause for the divergence and generate a patch (e.g., disable the emulation-detection code) for the malware. The iterative nature of this approach limits its scalability though. New traces and patches must be generated for each and every anti-emulation check.

2.3 Android

Android is a popular mobile system that is installed in millions of devices and accounted for more than 50% of all smart phone sales in the third quarter of 2011 [67]. It has been the target of most mobile malware [3] and recent research has shown that malicious applications exist in both the official and unofficial marketplaces with a rate of 0.02% and 0.2% respectively [68].

Figure 2.6 illustrates the architecture of the Android system from the perspective of a system programmer. At the lowest level, the Android system uses a customized Linux kernel to manage various system resources and hardware devices. System services, native applications and Apps (short for applications) run as Linux processes. In particular,

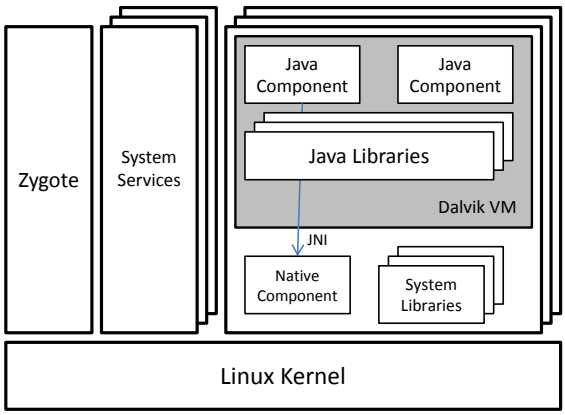

Fig. 2.6.: Overview of Android System

"Zygote" is the parent process for all Android Apps. Each App is assigned its own unique user ID (`uid`) at installation time and group IDs (`gids`) corresponding to requested permissions. These `uids` and `gids` are used to control access to system resources (e.g., network and file system) like on a normal Linux system.

All Apps can contain both Java and native components. Native components are simply shared libraries that are dynamically loaded at runtime. The *Dalvik virtual machine* (DVM), part of a shared library named *libdvm.so*, is then used to provide a Java-level abstraction for the App's Java components. At the same time, the Java Native Interface (JNI) is used to facilitate communications between the native and Java sides. As an aside, multiple layers of abstraction for Apps is not limited to Android. Windows Store applications have a similar structure. Those applications can be implemented in C++, C# or other languages in the .NET Framework [69]. C++ programs are compiled into native code while C# programs are interpretted using the Common Language Runtime (a virtual machine similar to JVM [70, 71]).

To create a Java component, an App developer first implements it in Java, compiles it into Java bytecode, and then converts it into Dalvik bytecode. The result is a Dalvik executable called a *dex* file. The developer can also compile native code into shared libraries (*.so* files) with JNI support. The dex file, the shared libraries and any other resources, including the *AndroidManifest.xml* file that describes the App and its requested permissions, are packaged together into an *apk* file for distribution.

For instance, DroidKungFu is a malicious puzzle game found in alternative marketplaces that has both native and Java components [72]. Its Java component exfiltrates sensitive information and awaits commands from the bot master. Its native component is used as a shell to execute those commands and it also includes three resource files that are encrypted exploits targeting known vulnerabilities - *adb setuid exhaustion* and *udev* [73] - in certain versions of Android.

Android Software Development Kit (SDK) Like Linux, there are Android ports for both the ARM and x86 ISAs. Android Apps that only have Java components do not have to worry about the underlying architecture since all Java code is interpreted using the Dalvik Virtual Machine. Native components must be compiled for use on the target ISA though.

The entire process of developing, targeting and testing Android applications is supported by the Android Software Development Kit or SDK [74]. Noteworthy components of the SDK are the Android emulator and prebuilt virtual Android devices. Cross compilers for compiling the native components into their respective target ISAs are included as well. The Android emulator is based on newer versions of QEMU that use the TCG. A virtual

Android device is a virtual machine configured with virtual sensor devices (e.g., GPS) and a build of the Android software platform. The prebuilt virtual devices represent generic phone configurations based on the ARM and x86 ISAs. A software developer can always create virtual Android devices with different profiles such as one for an ARM based tablet. Nonetheless, the Android emulator is used to emulate different virtual Android devices to facilitate application testing and debugging. In short, emulation based Android analysis is already provided as part of the SDK. What is missing is the ability to analyze malware.

2.3.1 Android Malware Analysis

Like malware analysis on the desktop environment, Android malware analysis techniques can fall into two categories: static and dynamic. For static analysis, the sample's dex file can be analyzed by itself or it can be disassembled and further decompiled into Java using tools like *dex2jar* and *ded* [75]. Standard static program analysis techniques (such as control and data flow analysis) can then be performed. As static analysis can give a complete picture, researchers have demonstrated this approach to be effective for many malware samples [68, 76].

However, static analysis is known to be vulnerable to code obfuscation techniques. In fact, the Android SDK includes a tool named Proguard [77] for obfuscating Apps and so obfuscated Apps should be common. Android malware may also generate or decrypt native components or Dalvik bytecode at runtime making static analysis of these dynamic components difficult. Indeed, DroidKungFu dynamically decrypts the exploit payloads and executes them to root the device. Moreover, researchers have demonstrated that bytecode

randomization techniques can be used to completely hide the internal logic of a Dalvik bytecode program [78]. Static analysis also falls short for exploit diagnosis, because a vulnerable runtime execution environment is needed to observe and analyze an exploit attack, and pinpoint the vulnerability. All in all, like for analyzing desktop malware, static analysis is insufficient by itself.

The Android SDK includes a set of tools, such as *adb* and *logcat*, to help developers debug their Apps. With JDWP (Java Debug Wire Protocol) support, the debugger can even exist outside of the device. However, just like how desktop malware detects and disables debuggers, malicious Android Apps can also detect the presence of these tools, and then either evade or disable the analysis. The fundamental reason is that the debugging components and malware reside in the same execution environment with the same privileges. Once again, virtualization based malware analysis has the advantage of isolation.

Despite the fact that Android is based on Linux, it is not straightforward to take the same desktop analysis approach used for Linux and apply it to Android malware due to the semantic gap problem. In Android, there are two levels of semantic information that must be rebuilt. In the lower level, Android is a Linux operating system where each App is encapsulated into a process. These processes can execute native code and this level is called the *native level*. Within each App process, a virtual machine (known as the Dalvik Virtual Machine) provides a runtime environment for the App's Java components. This second level is called the *Java level*.

Previous Virtual Machine Introspection research [17–19] only supports introspecting the native level but not the Java level components. Therefore, an emulation based dynamic analysis platform for Android malware must reconstruct semantic knowledge at two levels:

1) native level semantics that understand the activities of the malware process and its native components; and 2) Java level semantics that comprehend the behaviors in the Java components.

TaintDroid [79] and DroidBox [80] (based on TaintDroid) are examples of dynamic analysis tools for analyzing the Java components of Android Apps. TaintDroid is implemented as a modified Dalvik Virtual Machine. When installed onto a virtual Android device, it can be used to complement currently available emulation based analysis platforms for desktop malware. In this way, both the Java and native executions can be analyzed. The limitation is that they must be analyzed separately. Neither tool has information about the other and thus cannot collaborate. There are two approaches to surmount this limitation. Either a new messaging infrastructure is created between TaintDroid and the emulator or a single platform is introduced to not only bridge the two semantic gaps but also seamlessly bind the two views with the execution context so that malware with cooperating native and Java components can be analyzed at once.

Bouncer [81] is a recently announced dynamic analysis tool used by Google to test Android Apps prior to accepting them into the official Android Market. Oberheide et al. [82] determined that Bouncer is based on the Android emulator, but the details on what it is capable of (e.g., whether it seamlessly binds the Java and native contexts together for analysis) remains unknown. Bouncer is proprietary.

2.4 Taint Analysis

Taint analysis is one of the fundamental dynamic analysis techniques [6] and has been demonstrated to be crucial in many malware analysis projects [49, 51, 66, 83]. It is also known as *dynamic information flow tracking*, *taint tracking*, *definedness tracking*, and *data flow* and *control flow tracking*. The technique is based on *noninterference* [20] and has been used in many different research areas such as policy enforcement, exploit diagnosis, malware analysis, vulnerability analysis, test case generation [64], memory analysis [84, 85] and information flow quantification [86].

In taint analysis, data is labeled as either *tainted* or *untainted* and taint propagates from one data item to the next if information flows from a tainted source to the destination data item. Taint analysis has been implemented as a source translator [87–90] (e.g., compiler), a library [91–93] (e.g., Java String), in an instrumentation library [84, 94–97] (e.g., PIN), in a virtual machine or interpreter [79, 98–101] (e.g., Dalvik VM), in processor emulators [16, 56, 59, 102–104] (e.g., QEMU), and even in hardware [105–111]. Common to all of these taint propagation tools is the use of a taint propagation policy that governs what data is tainted, how the labels are represented and when data should be tainted. In other words, a taint propagation policy is a set of rules that define when and how taint should be propagated.

Taint propagation policies have been heavily researched. There are three common design considerations: *analysis-granularity*, *taint-granularity* and *special case support*. Analysis-granularity determines which type of operations propagates taint. One taint analysis implementation might propagate taint at the C statement level while another

propagates taint at the x86 instruction level. The advantage of the x86 instruction level implementation is it can track taint through all x86 programs (including those compiled from C programs); however, the disadvantage is the loss of high level semantic information (i.e., the semantic gap problem).

The range of analysis-granularity designs in the literature matches well with the different levels of programming languages. Researchers have defined propagation policies for high-level languages such as C [87] and Java [91–93, 99], scripting languages such as PHP [89], PERL [101] and JavaScript [90, 100], low level languages such as x86 assembly [56, 84, 94, 103, 104] and even at the gate level [108].

It is also common practice to implement taint propagation through an intermediate language [56, 59, 84, 94, 104] with simpler semantics and a reduced instruction set than through the language or instruction set (e.g., x86 and ARM) the IR emulates. The emulation code can range from a single instruction, to a basic block of instructions, to functions and beyond.

Taint-granularity determines the kind of data that is labeled. For example, in x86 tainting, data can be labeled at the 32-bit word-level, the byte-level or bit-level among others. While labeling taint at the 32-bit word-level will decrease label storage requirements, it is insufficient for distinguishing taints between the 32-bit `eax` register and the two 16-bit sub-registers `ah` and `al` that compose `eax`. In the literature, policies have been defined at the operand level [87, 105], 32-bit word-level [59, 111], byte-level [56, 59, 88, 95–97, 106, 107, 109, 110], and bit-level [84, 108].

Other taint analysis applications, such as the ones on Java String objects, apply different taint-granularities as well. Like how the `eax` register consists of the `ah` and `al`

sub-registers, Java Strings consist of UTF-16 characters. Consequently, Java String propagation policies have been defined at the character level [91, 92] and at the object level [93, 99], similar to the 16-bit versus 32-bit levels mentioned above.

Not all implemenations are equal. Some implemenations might have support for floating point operations while others do not or some might have special rules for propagating taint through the bit shift operations while otheres do not. These special considerations are instances of the special case support design parameter. Of particular interest is whether the taint propagation rules are *state aware*. State awareness is a measure of how much state information is used to determine the taint propagation rules. For example, the rules in a state agnostic taint tracker are functions of the operations - as defined by the analysis-granularity - only and not the operand values. The benefit of state agnostic policies is the rules can be defined off-line which in turn reduces the need to calculate taint at runtime, improving performance. However, the drawback is it can lead to false-positives. While dynamic taint analysis implementations are rarely state aware, many previously published trackers do implement special rules for handling special cases to reduce false-positives (see Section 5.5.2).

False-positives, *false-negatives*, *over-tainting*, *under-tainting*, *accuracy* and *precision* are different metrics that have been used to analyze and compare the effectiveness of different taint analysis platforms. These terms are informally defined below. It should be clear from the definitions that controlling false-positives and false-negatives is a goal of taint analysis. In this regard, previous research have only focused on reducing these metrics on a case-by-case basis using empirical studies. Therefore, a challenge of taint analysis is how to systematically control false-positives and false-negatives.

Over-tainting Given two taint propagation policies T_A and T_B, T_A *over-taints* T_B if there is a rule that propagates taint in T_A but not in T_B.

Under-tainting Given two taint propagation policies T_A and T_B, T_A *under-taints* T_B if there is a rule that propagates taint in T_B but not in T_A.

False-positive A rule is a *false-positive* if it propagates taint when taint should not be propagated.

False-negative A rule is a *false-negative* if it does not propagate taint when taint should be propagated.

Accurate A taint propagation policy is *accurate* if it does not contain any false-negatives.

Precise A taint propagation policy is *precise* if it is accurate and does not contain any false-positives (i.e., it does not contain false-positives or false-negatives).

2.4.1 Challenges

In addition to false-positives and false-negatives, two other major challenges of taint analysis are *sanitization* and *control-flow tainting* [64]. Sanitization uses specially defined rules to remove taint labels (i.e., label them as untainted) in certain special cases and is necessary to reduce over-tainting due to false-positives. These rules are either introduced for practicality (i.e., the analyst decides to ignore certain tainted data) or as means to reduce false-positives due to imprecise policies. Schwartz et al. [64] used the $b = a \oplus a$ statement that exclusive-ors a with itself and stores the result into b as a common example of sanitization for reducing false-positives. In the example, b should be sanitized even if a

```
1.  if (x == 0)        // 1a. if (x == 0)
2.    y = 0;           // 2a.   y = 0;
3.  else               // 3a. else
4.    y = 1;           // 4a.   y = 0;
```

Fig. 2.7.: Control-flow tainting example.

was labeled as tainted, because no matter what the value of a is, b will always be 0. Thus a policy that propagated taint from a to b will require an additional sanitization step to handle this special case. A state agnostic policy with a simple propagation rule that states "the output operand of $\oplus$ is tainted if either of the input operands are tainted" is an example of such a policy.

Control-flow tainting Information flows can be separated into *explicit flows* that result from data dependencies and *implicit flows* that result from control dependencies. Take the C source code in Figure 2.7 as an example. It is clear that the final value of `y` depends on the value of `x`, meaning there is information flow from `x` to `y`. Furthermore, the flow is implicit since there are no direct data dependencies between `x` and `y`. It is also clear that there is no information flow from `x` to `y` (not even implicit flow) in the commented source code since `y` always equals to 0, irrespective of the value of `x`. While the difference in information flow behavior between the un-commented and commented code are clear using static analysis, it is not clear during dynamic analysis.

Dynamic analysis is limited to the single path that is executed; there is incomplete information about the program. Hence, either statements 1 and 2 are executed or statements 1 and 4 are executed. If the "if" path (statements 1 and 2) was taken, then it remains unknown whether the "else" path contains statement 4 or 4a. Similarly, if the

"else" path was taken, then it remains unknown whether the "if" path contains statement 2, 2a or even 2b: `y = 1;`. All in all, control flow tainting is a fundamental limitation of dynamic taint analysis and has been proven. Volpano [112] proved that "there is no monitor-enforced policy that is sound and complete for secrecy." Soundness and completeness are measures of false-negatives and false-positives respectively, and dynamic taint analysis platforms are monitors that enforce the noninterference policy. Therefore, dynamic taint analysis will either have false-positives or have false-negatives.

Given this result, a number of taint analysis platforms that support control flow tainting relies on static analysis to reduce false-positives. Bao et al. [113] used static analysis to first identify "strict control dependence" relationships , and then used them to reduce false-positives due to control flow tainting. A method to approximate strict control dependence using dynamic instrumentation was also presented. Chang et al. [88] first conducted general data flow analysis on a program statically and used the results to direct information flow tracking at runtime. In DTA++, Kang et al. [104] used off-line analysis to identify "culprit" branches and limit control flow propagation to those branches. This increased the precision of implicit flow tracking as compared to Dytan [94] which used a purely dynamic approach.

This dissertation studies purely dynamic taint analysis, but it does not imply that control flow tainting is no longer a challenge. Control flow issues can still arise depending on the analysis-granularity. Take x86 level taint analysis for example. The `bsf` (Bit-Scan-Forward) x86 instruction is a simple ALU (Arithmetic Logic Unit) instruction that uses a while loop to iterate through the bit positions (0 to 31) of a 32-bit register to

find the position of the first '1' bit. This results in potential implicit flows that must be identified when defining the propagation rule for `bsf`.

Tainting and Noninterference While this dissertation focuses on monitors that enforce the noninterference policy (i.e., information flow), not all taint analysis platforms strictly adhere to this policy. Of the taint trackers surveyed in Chapter 5, nineteen enforce noninterference and only a handful use specially designed policies that do not. In Leakpoint, Clause et al. [85] defined a taint propagation policy for pointer arithmetic that is only loosely based on information flow. In empirical testing, Leakpoint was shown to be just as effective in identifying memory errors as Memcheck [84] which is based on information flow. Furthermore, researchers have used mixed taint- and analysis-granularities in a single implementation in an attempt to address over-tainting as a result of imprecise taint propagation rules. For example, Zhu et al. used a byte-level taint granularity for data in user-space and an object-level granularity for the same data in the kernel [96]. Chin and Wagner defined special propagation rules when dealing with Java String specific issues such as encoding and locales [92] and function summaries are widely used to summarize the taint propagation behavior within commonly used functions (e.g. libraries [56, 96]) and effectively sanitize taints. Slowinska and Bos discussed the sources of false-positives and false-negatives in pointer tainting [114].

2.4.2 Noninterference

The noninterference property was first described by Goguen and Meseguer [20] to analyze the information flows between users in a multi-user system. In the simple form of

noninterference used in this dissertation, there are two users in a shared system, *sender* and *receiver* that can issue commands. Since the system is shared, both the sender's commands and the receiver's commands can alter the system state. Thus, there is an opportunity for the sender to send information to the receiver through the shared system state. The noninterference property states that given an initial state and a sequence of commands that is an interleaving of the sender's and receiver's commands, sender is noninterfering with receiver if and only if the output seen by the receiver at the end of executing the sequence is the same if an alternate sequence that only contains the receivers commands was executed instead. In other words, the commands issued by the sender do not affect the output seen by the receiver. There is information flow from the sender to the receiver if and only if the sender interferes with the receiver.

Since taint tracking is designed to analyze the information flow between two data items and not users in a multi-user system, the noninterfering data problem can be mapped into a noninterfering users problem by coupling data-items to users. Analogously, given an initial state and a sequence of commands, there is information flow from the sending data item in the system state to the receiving data item in the system state if and only if changing the value of the sending data item results in a change in the value of the receiving data item after the sequence of commands are executed. It is the value of the data item that changed, not teh commands.

Design Considerations and Information Flow Analysis-granularity, taint-granularity and state awareness are the three common design considerations discussed in the previous section. The same parameters can be linked to the noninterference model above. A taint

propagation policy consists of a set of rules that enforce the noninterference property. Then, given a chosen analysis-granularity, the rules correspond to the different operations.

Using the x86 instruction granularity as an example, each x86 instruction will have a corresponding rule in the policy. Then, since the instructions are independent from each other, each rule will enforce the noninterference property. In other words the rules are in the form of "the output operand(s) is tainted if and only if there is information flow from a tainted input to the output operand(s)." This in turn means that sequence of commands consists of only the instruction itself. In the case of the $b = a \oplus a$ example above, the corresponding 32-bit x86 instruction is `xor dst,src`. There will be a rule that states "`dst` is tainted if and only if there is information flow from `src` to `dst` and `src` is tainted or from `dst` to `dst` and `dst` was tainted."

This rule assumes that the taint-granularity has been set at the 32-bit word-level which is not necessarily the case. If taint was labeled per byte, then there might be four rules of the form "the lowest byte of `dst` is tainted if an only if there is information flow from any tainted bytes of `src` to the lowest byte of `dst` or from any tainted bytes of `dst` to the lowest byte of `dst`." Similarly, none of these rules were functions of the initial state and are thus state agnostic rules. A state aware rule might state "if `src` and `dst` refer to the same data item then `dst` should be untainted, else follow the normal rules."

2.5 Summary

In summary, virtualization can be separated into emulation and hardware virtualization, where the latter has three distinct properties, efficiency, resource control and

equivalence. Implementing a dynamic malware analysis platform using virtualization benefits from the isolation property where it is difficult for the malware sample to disrupt analysis, even privileged kernel malware. The main disadvantage is the semantic gap problem where higher level abstractions, such as processes and threads, are lost. The semantic gaps can be bridged using virtual machine introspection techniques; however, the Android platform contains two levels of semantic information, native and Java, that must be rebuilt prior to analyzing Android malware.

Between emulation and hardware virtualization, the main advantages of emulation are flexibility and efficiency. As a result it is simple to implement an instruction tracer using emulation whereas it is more difficult using hardware virtualization. The main disadvantage is the lack of transparency.

Taint analysis is a fundamental dynamic analysis technique with three main challenges, false-positives and negatives, sanitization and tracking implicit flows. There are three common design parameters, taint-granularity, analysis-granularity and special case support that contribute to the problems.

3. MAKING EMULATORS TRANSPARENT

3.1 Introduction

The main advantages of emulation based dynamic binary analysis are flexibility and efficiency for code instrumentation, but the main disadvantage is the lack of transparency. It is extremely difficult (if not completely infeasible) to emulate every aspect of real hardware and thus malware can take advantage of these discrepancies to detect the emulated environment and stay dormant to avoid analysis. Researchers have investigated this problem extensively and identified a large number of different detection methods [10–12]. Furthermore, the measurement study in Section 3.5.2 shows that anti-emulation malware have become a prevalent threat in the wild.

To address the transparency issue, Dinaburg et al. used hardware virtualization to develope a system called Ether [60]. Since the malicious code is executed on real hardware, this approach can achieve ideal transparency. However, Ether is not the ultimate solution. Being a hardware virtualization based analysis platform, it lacks the benefits of efficiency and flexibility.

This chapter proposes precise heterogeneous record and replay (PHRR) as an alternative method for addressing the transparency issue. The method involves recording malware execution using hardware virtualization and then replaying the executing on an

emulation based malware analysis platform so as to attain the advantages of flexibility, efficiency and transparency.

Regular record and replay systems targeted at the execution of a single user-level process [115–119] and a whole virtual machine [120, 121] can be found in the literature. However, they cannot be directly used for malware analysis. In most cases, record and replay take place on the same type of system. In the case of emulator based record and emulator based replay, transparency is still an issue. Alternatively, if hardware virtualization is used for both record and replay, then the advantages of emulation based analysis are lost.

The idea of heterogeneous replay was first proposed and implemented in Aftersight [122], which records virtual machine execution from VMware and replays it in QEMU for heavyweight analyses (such as bug detection) on production workloads. In contrast to Aftersight, the record and replay method presented in this Chapter needs to work under the malicious context: malware tries to detect every possible heterogeneous property between the recorder and replayer.

One challenge of precise heterogeneous replay is in striking a balance between the recorder and the replayer. On one hand, if the recorder does not record enough events and states, the replayer cannot precisely reconstruct the execution. On the other hand, if the recorder gathers complete information for every single instruction or event and leave an easy task to the replayer, the recording performance would degrade. To strike the right balance, various operations and instructions are classified into different categories according to the technique used to ensure precise replay without sacrificing too much performance.

Precise heterogeneous record and replay was implemented in a prototype system called *V2E*. The recorder has been implemented in KVM [13], and TEMU (a dynamic binary analysis platform [15]) has been modified to precisely replay the execution. With minor changes, the existing analysis plugins (such as taint analysis, unpacker, and tracing) work as designed, achieving the advantages of transparency and greater analysis efficiency. Since analysis is separated from execution, the same scheme can be applied to other emulation based binary analysis platforms as well. Recording is not limited to KVM and replaying is not limited to TEMU.

To determine the effectiveness of PHRR, V2E was evaluated using both synthetic and real world anti-emulation malware samples. These same samples were successfully recorded and replayed on the modified TEMU revealing behavior hidden from the original TEMU platform.

The rest of the chapter is organized as follows. The next section lists the design goals that are essential for in-depth malware analysis and gives an overview of the approach. Section 3.3 and Section 3.4 describe the design and implementation of the recording component and replay component respectively. Section 3.5 presents the experimental results. Section 3.6 discusses the limitations of the current implementation. Finally, some intermediary conclusions are drawn in Section 3.7.

3.2 Design Goals & Approach

The design goals for in-depth malware analysis and approach to address them are presented in this section.

3.2.1 Design Goals

As discussed previously, the following design goals are essential for in-depth malware analysis:

- **G1: Transparency.** The presence of the analysis environment should remain invisible to malware, voiding its intent to escape investigation. This will be provided by precise heterogeneous record and replay.

- **G2: Flexibility.** It should be relatively simple to add custom instrumentation on malicious code execution. In many cases, this instrumentation can be heavyweight, such as dynamic taint analysis and instruction-level tracing. This is a characteristic of emulation based analysis platforms.

- **G3: Efficiency.** The efficiency for malware analysis is two-fold: 1) it should be efficient enough to monitor computation intensive and highly interactive malware; and 2) performance overhead for heavy code instrumentation should be acceptable. This is another advantage of emulation based analysis.

- **G4: Adjustable View.** A benefit of record and replay is one can be selective about what to record and therefore control what is replayed and analyzed. This can improve performance. Given that malware can present itself in various forms, such as user process, shared library, dynamically injected code, kernel module, etc., this goal expresses the desire to be able to adjust the analysis focus to concentrate on malware's behavior instead of the execution of the rest of the system.

3.2.2 Architecture

The overall architecture is depicted in Figure 3.1. The malware sample under investigation is loaded into and executed in the guest system using hardware virtualization to achieve transparency (**G1**). Although hardware virtualization may still be detected under certain circumstances [11] and a remote time source can be used to measure real timing differences [123], as hardware virtualization has been widely deployed on production systems, detecting hardware virtualization environments becomes increasingly irrelevant. Ether demonstrated that hardware virtualization can achieve transparency in a practical sense [60].

The guest system is partitioned into two realms or domains. The malware resides in the *recording realm* and the rest of the guest system (such as the guest OS and the other applications) remains in the *main realm*. Depending on where the malware is present, a user process, a shared library, a kernel module, or any combination of them can be put into the recording realm for a closer look at the malware's behavior. Such a separation fulfills the adjustable view design goal (**G4**). It also partially addresses the efficiency issue (**G3**) because irrelevant system execution is excluded from the recording realm and consequently from the log. Some analysis techniques (such as Panorama [56] and HookFinder [54]) do need to observe the entire system execution. In this case, the recording realm includes the entire guest system, falling back to whole-system recording. The design is not optimized for whole-system recording though and this support is left as future work.

The log obtained from the recorder is fed into the replayer. Using dynamic binary translation, the replayer is able to offer acceptable performance for fine-grained code

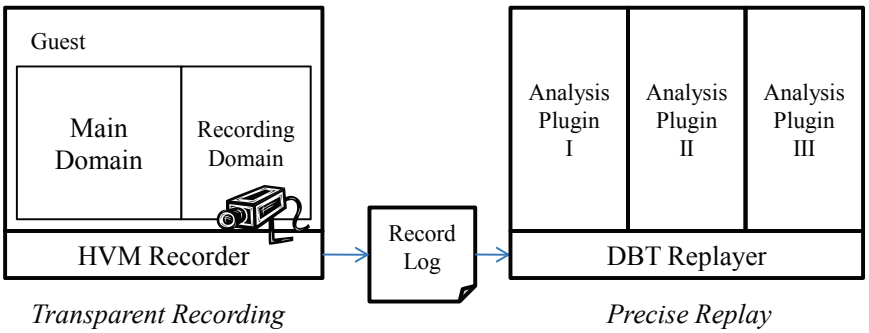

Fig. 3.1.: Architecture Overview

instrumentation and thus achieves analysis efficiency (the second part of **G3**). The replayer design facilitates any existing malware analysis platforms that are based on dynamic binary translation. Therefore, the existing analysis plugins on these analysis platforms can continue to work with minimum changes. It addresses the instrumentation support goal (**G2**).

3.2.3 Precise Heterogeneous Record and Replay

The claim is: malware execution can be recorded in a transparent and efficient manner using hardware virtualization, and the recorded execution can be precisely replayed using dynamic binary translation. That is, at every single execution time-step, the program state in replay is exactly the same as in recording in spite of the fact that the malware execution is trying to detect various discrepancies between real hardware and the emulated system.

Semi-Formal Definition Let S be the set of all program states (including CPU registers and memory). Then, at each time t, S_t represents the state at that time and S_0 is

the initial state. Let I be the set of all possible inputs and thus I_t specifies the input at time t and may be *null* to indicate no input occurs at that time.

A transition function $f : S \times I \rightarrow S$ is used to characterize the real hardware machine: $S_t = f(S_{t-1}, I_{t-1})$. Similarly, there is a transition function f' for the emulated machine: $S'_t = f'(S_{t-1}, I_{t-1})$. Suppose that $f' = f$, then given S_0 and the inputs I the whole execution can be replayed precisely as recorded. However, according to automata theory, determining if $f' = f$ is equivalent to the problem of determining whether two Turing machines are equal, which is known to be undecidable [124]. It then follows that in practice, they are assumed not to be equal, that is $f' \neq f$, because it is nearly impossible to correctly emulate some aspects of hardware.

Given that the transition functions are different, the states S_t and S'_t are also expected to be different given the same input and previous state. Therefore, in addition to recording S_0 and inputs I, for any moment in time u where $S_u \neq S'_u$, the state change $\Delta_u = S_u - S_{u-1}$ is recorded as well. Note that the state change is not defined as $\Delta_u = S_u - S'_u$ because recording and replaying are done separately. This means that it is not necessarily possible to calculate S'_u while the program's execution is being recorded.

The new transition function f'_r is defined as:

$$S'_t = f'_r(S'_{t-1}, I_{t-1}, \Delta_t) = \begin{cases} S'_{t-1} + \Delta_t & \text{if } \Delta_t \neq null \\ f'(S'_{t-1}, I_{t-1}) & \text{Otherwise} \end{cases}$$

In other words, during replay, whenever a state change Δ_t has been recorded for time i, the state is directly updated such that $S'_t = S'_{t-1} + \Delta_t$. There is no need to apply the transition

function. It is important to note that the references to S'_{t-1} can be replaced by S_{t-1} since they are equal. The claim is: with (S_0, I, Δ) and f'_r, $S'_t = S_t$ always holds true for all time t until the program ends. The induction proof is outlined below.

Basis: The base case is clear since $S'_0 = S_0$.

Induction: For the induction case, assume that the relationship holds at time t: $S'_t = S_t$. It remains to be shown that $S'_{t+1} = S_{t+1}$. There are two cases. In the first case where $\Delta_{t+1} \neq null$, $S'_{t+1} = S'_t + \Delta_{t+1}$, which can be rewritten as $S'_{t+1} = S'_t + S_{t+1} - S_t$. Given that $S'_t = S_t$, $S'_{t+1} = S_{t+1}$. In the second case where $\Delta_{t+1} = null$, then by definition of f'_r, $S'_{t+1} = f'(S'_t, I_t)$. By designing the recorder such that $\Delta_{t+1} = null$ implies $f(S_t, I_t) = f'(S_t, T_t)$, $S_{t+1} = S'_{t+1}$. Details on how this can be achieved are presented next.

From Theory to Practice. The previous discussion shows that the key to successful record and replay is to determine when to use f' and when to apply Δ. In other words, if one is confident that certain instructions and events can be correctly emulated in software, then simply emulate them. Otherwise, the state changes should be recorded and then applied during replay.

Fortunately, for general instructions like data transfer (e.g., `mov`, `push`, `pop`), control transfer (e.g., `call`, `ret`, `jz`, `jmp`), and integer arithmetic (e.g., `add`, `shl`), it is fairly easy to emulate them correctly in software because their semantics are simple and remain the same across processor series.[1] Moreover, these instructions are the vast majority in program execution. As a result, the efficiency of both recorder and replayer can be ensured.

[1]Note that these common instructions may still cause discrepancies in exceptions, which are handled separately.

This is a valid assumption for the mature QEMU emulator, because these common instructions are tested over and over again in many different application contexts.

External interrupts, memory-mapped I/O (MMIO), port I/O, direct memory access (DMA), and timestamp counter are inputs I to the guest system. Like other deterministic replay systems [118, 120, 122], these events are recorded if they occur in the recording realm only.

Software exceptions, model-specific registers, and the `cpuid` instruction are not generally treated as inputs in previous replay systems. However, it is extremely difficult to emulate them correctly. Software exceptions are triggered when certain condition checks fail in the processor. It is fairly complicated to emulate all these condition checks in the exact same way as in the real processor, not to mention that a specific processor may have CPU bugs that raise incorrect exceptions [9, 10]. The behaviors of model-specific registers and the `cpuid` instruction are processor specific as well. Overall, it is a daunting task to correctly emulate all the specifics of just the common CPU series. Therefore, exceptions, model-specific registers and `cpuid` are recorded as state changes Δ.

Floating point instructions and SIMD (Single Instruction Multiple Data) instructions (e.g., MMX and SSE) are generally difficult to emulate correctly as well. It is possible to also record the results of these instructions as Δ. However, for programs that heavily perform these operations, the performance for both record and replay may greatly degrade (one Δ event is needed per instruction). Alternatively, these instructions are directly passed to the hardware processor during replay. This solution assumes that the replayer is running on a machine supporting the same set of floating point and SIMD instructions.

Operation Type	**Solution**
Data Transfer / Control Transfer / Integer Arithmetic	Emulate
Interrupts / MMIO / Port I/O / DMA / TSC	Replay as I
Exceptions / System Registers / CPUID	Replay as Δ
Floating Point / SIMD Instructions	Pass through

Table 3.1: Operations and Corresponding Solutions.

This assumption can easily hold by running the recorder and the replayer on the same kind of machines or even the same machine.

As a summary, Table 3.1 lists the special operations and their corresponding solutions: *emulate*, *replay*, or *pass-through*. It needs to be emphasized that a platform following this design principle does not immediately become completely transparent. Emulation bugs or missing inputs and state changes cannot be precluded. Once identified though, these bugs can be fixed and the transparency of the platform will be further improved.

3.3 Transparent Recorder

For successful replay, S_0, I and Δ need to be captured in a transparent and efficient manner. How this is goal is achieved using hardware virtualization is described in this section.

3.3.1 Mediating Recording Realm

In order to monitor malware in various forms, including kernel modules, shared libraries and processes, the recording realm is defined at the page-level granularity and the interaction between the recording realm and the rest of the system needs to be mediated.

In hardware virtualization, Two Dimensional Paging (TDP) is a memory virtualization mechanism. While the conventional page table pointed by CR3 in the guest is used to translate a Guest Virtual Address (GVA) into its Guest Physical Address (GPA), the second-layer page table maintained by the hypervisor translates the Guest Physical Address into the Host Physical Address (HPA). The maintenance of the second-layer page table is invisible to the guest and is used to mediate the recording realm.

Specifically, two TDP tables are created to partition the guest physical memory into two memory spaces, one for the recording realm and the other for the rest of the guest system - the main realm. The code pages that belong to the monitored malware will be loaded into the recording realm, such that the interactions between the monitored malware and the rest of the system can be mediated by TDP page faults and other VMExit (transitions from the guest to the hypervisor) events. These VMExit events are invisible to the guest system.

This TDP-based recorder design is flexible enough to monitor a small code module, a full user process, and even the entire guest system, depending on what pages are loaded into the recording realm.

3.3.2 Basic Scheme

In the basic design, the two guest physical memory spaces are mutually exclusive. That is, each individual guest physical page can only be present in either the recording realm or the main realm, but not both. This basic design ensures mediating all the inputs and events for the recording realm is simple.

```
1. int adore_root_filldir(void *buf,
      char *name, int nlen, loff_t off,
      ino_t ino, unsigned x)
2. {
3.   struct inode *inode = NULL;
4.   int r = 0;
5.   uid_t uid;
6.   gid_t gid;
7.
8.   if ((inode=iget(
           root_sb[current->pid% 1024],
           ino)) == NULL)
9.     return 0;
     //lines 10 to 20 are
           omitted for brevity
21. }
```

(a) C source

```
d88888550 <adore_root_filldir>:
550:  push  %ebp
 551...56C: //set up stack,
           //%eax = current @L8
56C:  mov   0x6c(%eax),%edx
            //%edx=pid @L8
56F:  xor   %edi,%edi
571:  test  %edx,%edx
573:  mov   %edx,%eax
575:  jns   57d <adore_root_filldir+0x2d>
577:  lea   0x3ff(%edx),%eax
57D:  push  $0x0
57F:  push  $0x0
581:  and   $0xfffffc00,%eax
586:  sub   %eax,%edx
588:  pushl 0x1c(%ebp) //push ino @L8
58B:  pushl x0(,%edx,4)
              //push root_sb[...] @L8
              // call iget @ line 8
592:  call  593 <adore_root_filldir+0x43>
   //The rest is omitted for brevity.
```

(b) disassembly

Fig. 3.2.: adore_root_filldir

A simplified *adore-ng* rootkit [125] is used as an example to illustrate this basic scheme. The C source code and the corresponding disassembly are shown in Figure 3.2. In brief, the original pointer to `root_filldir` has been replaced by a pointer to `adore_root_filldir` to hide certain files. Suppose the execution of this kernel rootkit is to be recorded. Initially the code page of this kernel module is moved from the main realm to the recording realm. It may be treated as S_0. Figure 3.3 (A) illustrates this situation.

When the guest system is about to call `root_filldir`, the execution is redirected to `adore_root_filldir`, with virtual address 0xd88888550 and physical address 0x16876550. Since the physical page 0x16876000 is not present in the main realm any more, this control flow transition will trigger a TDP page fault which results in a VMExit. The recorder located in the hypervisor captures this TDP page fault and switches the memory space to the recording realm, which is shown in Figure 3.3 (B). In addition, the current CPU state (all the registers and flags) is recorded, as input I_0.

On fetching the first instruction in `adore_root_filldir`, two more TDP page faults are triggered for the page table directory page (PD) and the page table entry page (PTc) respectively. This is because the TLB has been flushed during the realm switch, and the CPU needs to look up the physical address of the first instruction using the page tables. These two pages are also moved into the recording realm and their contents recorded as another input. This is the desired behavior, because the replayer will need the page table pages for address translation. Moreover, by including the page table pages, the problem of page swapping and re-mapping in the guest is automatically handled in both recorder and replayer. Figure 3.3 (C) shows this moment.

This first instruction (`push %ebp`) writes onto the stack. As the stack page is absent in the recording realm, this operation triggers TDP page faults for the stack page (MS) and the corresponding page table pages (PD and PTs). as shown in Figure 3.3 (D). In this example, PD has already been moved into the recording realm, so no TDP page fault happens for PD.

Then the execution continues without causing any VMExits until 0xd888856c. This instruction (`mov 0x6c(%eax), %edx`) reads from a data page (D), which is not present in the recording realm. Similarly, this data page (D) and its corresponding page table page (PTd) are moved into the recording realm and recorded (see Figure 3.3 (E)).

The execution further proceeds to the instruction located at 0x169f7592. It calls `iget`, a kernel function. A TDP page fault is raised because the jump target is absent in the recording realm. The faulting EIP shows that it does not belong to the malware module. So, the recorder switches back to the main realm (see Figure 3.3 (F)). In addition, a

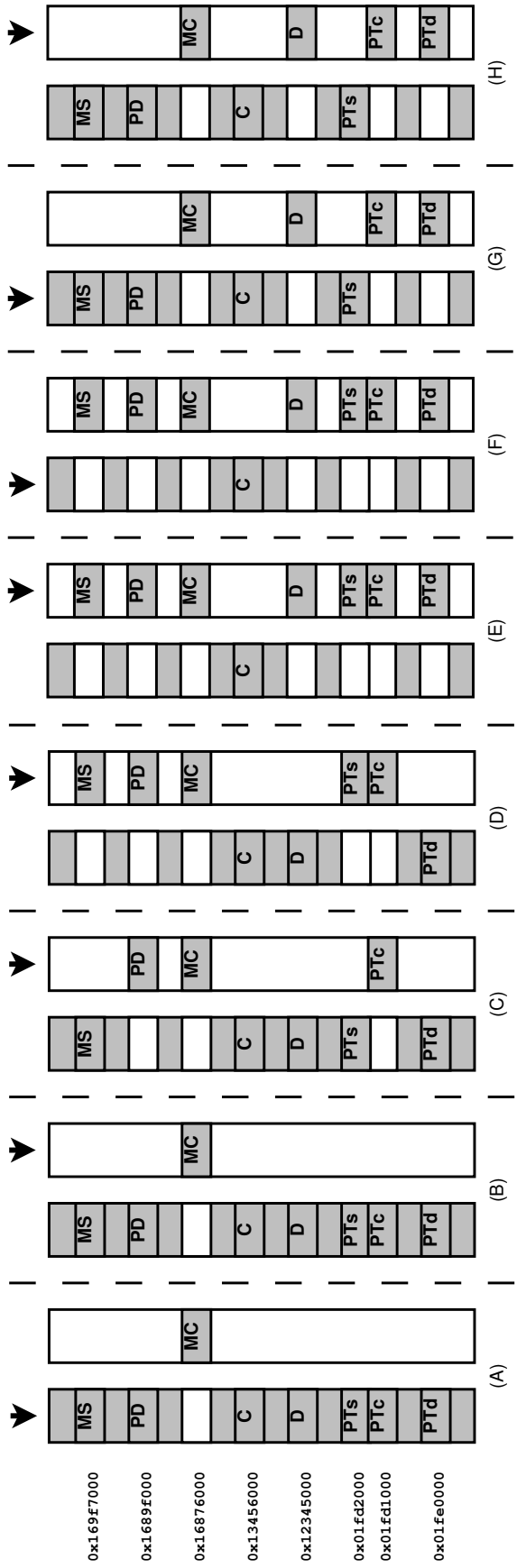

Fig. 3.3.: **TDP snapshots for adore_root_fill.** The two columns represent two guest physical memory spaces for the main realm and the recording realm respectively. A shaded block represents a present page, while a blank block indicates an absent page. The arrow on top signifies which realm is active.

"JumpOut" event is recorded at this point, indicating that the execution has transferred out of the recording realm.

The `iget` kernel function now resumes its execution in the main realm. While it accesses the parameters, another TDP page fault occurs because the stack page (MS) has been moved to the recording realm. So the stack page (MS) and the corresponding page table pages (PD and PTs) are moved back into the main realm. This behavior is also desired, because the next time when the recording realm reads one of these pages, it will be captured and the new page content recorded as a new input. Figure 3.3 (G) illustrates this situation.

When `iget` finishes and returns, a TDP page fault occurs because the jump target is not present in the main realm. Thus, the memory space is switched back to the recording realm, which is shown in Figure 3.3 (H). The CPU state is recorded and execution resumes in the recording realm. The subsequent execution of `adore_root_filldir` follows a similar cycle.

3.3.3 Other Inputs

The previous example only shows how to capture inputs from CPU states and memory. Other kinds of inputs need to be handled as well. To handle control transitions such as interrupts and exceptions using the same basic scheme, the Interrupt Descriptor Table (IDT) is prevented from being present in the recording realm. Any interrupt or exception will force a lookup into the IDT which will in turn trigger a TDP page fault. This fault is treated as a control transition and a new CPU state is recorded when the execution returns

to the recording realm. When executing in the recording realm, instructions like `cpuid`, `rdmsr`, `in` and `rdtsc` need to be recorded as inputs I and state changes Δ as well. With hardware virtualization support, the VMCS is configured to trap these instructions back into the hypervisor where their results are recorded. DMA transfers may change memory pages in the recording realm without CPU intervention. When DMA writes into a page resident in the recording realm, that page needs to be recorded as a new input. The DMA controller is emulated in software. So this DMA write can be intercepted and recorded as input.

3.3.4 Optimizations

The basic scheme enforces two mutually exclusive realms. In many cases, this is unnecessarily expensive. If the two realms alternately reads a shared page, then the basic scheme will repeatedly remove that page from the recording realm, and later move it back and record it even if the page contents have not changed. Several optimizations are employed to allow these two realms to share pages.

Sharing Data Pages. To enable sharing of data pages, a "Remove-On-Write" (ROW) principle, which is similar to Copy-On-Write, is used. More specifically, the two realms are allowed to share pages, but these pages are set to be read-only. When one realm attempts to write to a page, a TDP write violation will be triggered and that page is then removed from the other realm. This optimization works especially well for the page table pages, because both realms need these pages for address translation, and these pages seldom change.

Sharing Code Pages. When recording a full process, a problem that can arise is when both realms need to access shared library code pages. These code pages will be moved back and forth between the main realm and the recording realm, according to the basic scheme. Similar to sharing data pages, these code pages are present in the both realms and are initially marked to be read and execute only. While it improves performance, this optimization can disrupt the realm transition detection logic outlined above during context switches. That is, if shared library pages are executable in both the main and recording realms, the kernel is executing in the main realm (process level recording), there is a context switch to the recorded process and the process resumes at one of the shared libraries, then the TDP page fault will not occur. This results in the monitored program executing in the main and not the recording realm. To prevent this problem, the NX (Non-Executable) bits for these code pages are manipulated to capture the moment when execution transitions into these code pages in the monitored process.

More specifically, context switches are monitored by intercepting CR3 writes (once again by configuring the VMCS). When the execution context switches to the recorded process, the main realm is switched to the recording realm as normal. However, there is a gap between this context switch and the user-level execution, because the context switch is performed in the kernel space and the execution will continue in the kernel space for a while before it transitions to the user space. In order to capture the entry point to the user space, all of the pages in the recording realm are marked as Non-Executable. Although the kernel execution will trigger TDP page faults and in turn these pages will be loaded into the recording realm, these pages are not recorded. Essentially nothing is recorded until a TDP execute violation with the faulting EIP in the user space is detected. This is the

entry point back into the recorded program. The pages loaded during the kernel execution are removed so transitions back into the kernel can be detected later. The pages marked NX will be restored as well.

3.3.5 Bridging the Semantic Gap

Analysts usually specify which malware to monitor by its executable name, whereas the recording realm operates directly on guest physical pages. Therefore,there exists a semantic gap, which is bridged using VMI. More specifically, system calls are intercepted and kernel data structures in the guest system are parsed to extract the OS-level semantics, such as the process list and the module memory map. By this way, the process name is mapped to the corresponding CR3 value, and module names to their virtual memory ranges. Then guest virtual addresses can be translated into guest physical addresses using the guest page tables.

The mapping from guest virtual to guest physical address may change over time due to page swapping. The newly mapped physical page will be captured and recorded when it is accessed later, but the physical page that is no longer mapped needs to be removed from the recording realm immediately. To do so, page table changes that affect the pages in the recording realm are captured and recorded. According to the data page sharing mechanism, the page table pages associated to the recording realm are shared in both realms and set to be read-only. Therefore, any changes to these page table pages will be trapped to the hypervisor. Checking which page table entry has been modified reveals which guest physical page needs to be removed from the recording realm.

3.3.6 Shadow Time Stamp Counter

As extra TDP page faults and other VMExits are needed for recording malware execution, malware may detect the underlying recording behavior by examining the advance of the Time Stamp Counter (TSC). This can be done by using the `rdtsc` instruction to read the TSC model-specific register. A shadow TSC is used to hide this artifact.

The shadow TSC is an estimate of how much time the guest actually runs. It is not perfect. It is calculated as follows: Let t_i be the value of the host TSC before `vmresume` is executed. Let t_i be the value of the host TSC right before the transition into non-root mode, and t_o be the value of the host TSC right after the CPU returns to the host. Then, let t_e and t_x be the time it takes to enter the guest and exit to the host and t_g be the actual execution time for the guest, then $t_o - t_i = t_e + t_x + t_g$. To approximate $t_e + t_x$, the VMCS is configured to enable `rdtsc` exiting (i.e, a VMExit occurs whenever the guest executes `rdtsc`. A guest program that loops `rdtsc` is then executed to obtain the average of $t_e + t_x$.

This estimate of the total entry and exit times is then used to calculate the time spent executing the guest while recording. More specifically, t_i and t_o are captured and t_g is calculated as $t_g = t_o - t_i - (t_e + t_x)$. t_g is added tot he shadow TSC, which is returned to the guest whenever the guest queries the TSC. Due to unnecessary TLB flushes when transitioning between the guest and host modes, $t_e + t_x$ must be adjusted to account for TLB misses. This includes the misses due to the page table, code and data pages access, plus interrupts. Effectively, there are different $t_e + t_x$ averages for the different conditions.

3.3.7 Implementation

The recording component is implemented in the KVM module in Linux Kernel version 2.6.32. The code base of KVM is well organized. While `vmx.c` and `svm.c` contain the hardware specific code for Intel and AMD virtualization extensions respectively, `mmu.c` contains the memory management unit code that is common to both architectures. Within `mmu.c` is the `tdp_page_fault` function that is called by both VMX and SVM, and is where the realm control and enforcement logic is implemented. All memory based inputs are handled at this location. All non-memory based inputs to the recording realm are handled in the architecture specific implementation files.

Memory Management. KVM uses MMU-notifiers to learn of changes to the host process' (e.g., QEMU's) page table. Once a change is detected, KVM determines whether the old physical page (e.g., the one that has been swapped out) is pointed to by the current TDP page table and if so, make the necessary changes. Similarly, the recorder registers its own MMU-notifiers so changes to the page table are reflected in both the main and recording realm TDP page tables.

Logging. Being a kernel module, KVM should not directly write to files. To enable logging, a user-level program is used to commit the changes to a log file. KVM and the user-level program communicate using a shared memory queue that is mapped into the user-level program's virtual memory space. More specifically, the recorder exposes a filed-based interface through the "/proc" file system. The user-level program opens the file and writes to and reads from it to send and receive messages to and from the recorder.

Given a new log entry (e.g., a new page input), the recorder will first copy the contents into the shared queue in the user-level program's memory space and then send a message to the user-level program using the file interface. Since file reads and writes by the user-level program are blocking, the file based scheme is self-synchronizing.

In V2E, a page that has been loaded into the recording realm is fully and completely recorded. Obviously, this is only necessary when the page is recorded for the first time. When the page is modified in the main realm and loaded back again, it is possible that only a small portion of the page has changed (e.g., the stack should not change much through function calls). At first glance, a simple optimization such as recording only the differences or "diff" of the old and new pages is desireable, however, evaluations showed that this is unnecessary. The I/O bandwidth available on the test system was not saturated. Thus, to improve performance, the recorder does not calculate the differences at runtime. The log file is simply compressed after the fact to reduce storage requirements.

Event Landmark. Synchronization is an important aspect of record and replay systems. Nondeterministic events, such as interrupts, must be replayed at exactly the same moment in the program's execution during replay otherwise the two executions will diverge. The problem is exacerbated in precise heterogeneous record and replay, because previously deterministic events are now nondeterministic. Take exceptions as an example. In PHRR, the replayer is required to exhibit the same exception behavior as the recorder. If there is an error where the emulator does not throw an exception while real hardware does, the recorded exception must still be replayed at the right moment. In effect, this exception is nondeterministic.

Previous systems used the branch counter as a landmark. The branch counter increments when a branch instruction is committed (i.e., actually taken and not speculatively taken). By recording the number of branches committed thus far and the current EIP value for an important event that must be replayed (e.g., an external interrupt), the replayer can replay the same event at the same point during the program's execution.

In V2E, only the CPU state (including the EIP, registers and flags) is used as the landmark. It is a simple scheme and is not as accurate as the branch counter, because two execution points may happen to have the same CPU state. It has been sufficient in practice though. There are a large number of events (e.g., memory accesses and control transitions from and to the recording realm) that need to be recorded in precise heterogeneous record and replay. Each event serves as a synchronization point and so the role of the landmark not as important as in the other replay systems that synchronizes more seldomly. To put it differently, the landmark only needs to be accurate between two synchronization points, because the synchronization points themselves serve as landmarks. In the end, branch counter based landmarks is left as future work.

3.4 Precise Replayer

The purpose of the replayer is twofold: precisely replay the execution and events as recorded using hardware virtualization, and support emulation based malware analysis.

3.4.1 Dynamic Binary Translation and QEMU

In brief, QEMU uses dynamic binary translation to emulate one instruction set architecture (e.g., x86) on top of another (e.g., ARM). In DBT, a block of guest code is translated prior to being executed. The translated blocks can be cached for efficiency. The hardware memory management unit is emulated in software as is the translation look aside buffer.

While QEMU uses different techniques, such as lazy flags calculation, to improve efficiency, the same techniques can be used for emulation detection. The recorder was designed to capture all of the important events during the program's execution and the replayer must not only replay the events precisely as recorded, but some of the optimizations must also be disabled to facilitate precise replay.

3.4.2 Changes for Precise Replay

Considering the challenges in software emulation and dynamic binary translation, several design changes in the work flow of software emulation are used to ensure precise replay. Particularly, the dynamic binary translator in QEMU is modified to comply with the following design changes.

New Translation Logic. Instructions are classified into three categories during dynamic translation: general-purpose, FPU, and others. General-purpose instructions include data transfer, control transfer, and integer arithmetic. They are translated according to their simple semantics. To avoid discrepancies in flag calculations, lazy flag calculation is

disabled. That is, the EFLAGS register is immediately calculated after each instruction. Since exceptions are recorded and will be replayed, the logic for checking and raising exceptions is unnecessary and is completely removed except for page faults. Page faults are used for synchronization.

Floating point and SIMD instructions execute directly on the hardware FPU. To ensure correctness, these instructions are translated into wrapper functions that pass the operations directly to the real FPU using the state from the emulated virtual machine. For example, in QEMU's software emulation approach, a floating point instruction `fadd %st1, %st0` would be translated to call a helper function `helper_fadd_ST0_STN` to emulate this instruction in software. In the pass through approach, a piece of assembly code (`__asm__("fadd %st(1), %st(0)")`) is directly inserted instead of the call to the helper function. As this instruction takes two FPU registers, the instruction can be passed directly to the FPU. On the other hand if the instruction takes any operands from memory or the general-purpose registers, the operands need to be copied from the guest environment to the host and vice versa. For example, the instruction `fadds %0xc(%ebp)` adds a memory operand with `st(0)` and stores the result back into `st(0)`. Since this memory operand is located in guest memory, it is first copied into a temporary location on the host before the floating point operation executes natively on the host. This behavior is shown in the following code snippet where `ldl(A0)` is a function that returns the contents of the guest memory located at the GVA within A0 (i.e., `A0 = %0xc(%ebp)` in the example).

```
unsigned long temp = ldl(A0);
__asm__("fadds %0;" : : "m" (float)temp));
```

These natively executed FPU instructions may raise exceptions - host exceptions and not guest exception - as well. To prevent the exceptions from disrupting the host QEMU process, exception handlers are registered to catch and ignore them. Any exceptions that should be handled by the guest would have been recorded. The goal of the replayer is to replay the events, including exceptions, precisely as dictated by the log.

All remaining instructions are translated into "nop", expecting that the results of these instructions are correctly replayed from the log. That is, no translated code will be generated except to advance the program counter to the next instruction.

To address the page boundary issue described in Section 2.1.2, the DBT translation logic is altered slightly. Translation never crosses the page boundary. If the program counter crosses a page boundary during translation, then the current code block is ended and the instruction starts the next block. In this manner, the first instruction of the next block is guaranteed to be the instruction that crosses the page boundary. As a result, the page fault that results due to DBT (emulation) and the page fault that results during instruction fetch (real hardware) occur at the same location. Note that this increments the branch counter and needs to be taken into account if the branch counter based event landmark is used.

Replay Logic. As page-level recording is based on TDP, the same second level page table mechanism is needed in software emulation to correctly replay logged events on demand. A *physical page container* is introduced for this purpose. This physical page container indicates if a physical page has been loaded from the log and thus is present. In essence, the physical page container replicates the TDP page table of the recording realm

during replay. When the replayed execution accesses a page that is absent in the physical page container, it implies that there was a TDP page fault during recording. Consequently, the missing memory page is loaded from the log and the CPU state updated at the right moment.

In addition, the current CPU state is compared with the landmark of the next log event at the end of every instruction. If the landmark matches, the logged event is replayed. This event may be a control transition caused by interrupts or exceptions, a state change made by special-purpose instructions, or a realm change. These were captured as "JumpOut" events.

3.4.3 Example Walk-through

The same *adore-ng* example is used to walk through the replay logic. As the first log event, the code page MC is loaded in the physical page container. This initial state is the same as that of the right column in Figure 3.3(A).

Then the second event is the CPU state for the entry point of `adore_root_filldir`. The replayer updates the CPU state accordingly. The code block starting with the EIP at 0xd8888550 needs to be translated and put into the translated code cache before it is executed. This translation triggers a page table lookup to translate the EIP virtual address into a physical address. Consequently, the page table pages (PD and PTm) are loaded from the log on demand, because they are not present in the physical page container.

When the translated code executes, the first instruction pushes onto the stack. At this moment, the page fault triggers the page table page PTs to be loaded from the log during

address translation and then the stack page MS is loaded for the memory write. Similarly, the page table page PTd and the data page D are loaded at the right moments.

At the end of the call instruction at 0xd8888592, which jumps to the kernel function `iget`, a control transition happens. This JumpOut log event is followed by several events for removing pages (MS, PD, PTs) and culminates with a CPU update event. The page removal events in conjunction with the CPU update event represents the execution of `iget` in the main realm, which has been skipped. Consequently, the current program state is that of the instruction at 0xd888885a7, when `iget` just returned back into the recording realm (the right column of Figure 3.3 (H)). As this point, the software TLB is also flushed, because changes may have been made to the page table during the skipped execution.

Replay continues until all entries in the log are consumed. As this example describes the basic scheme, quite a few pages (such as page table pages) are removed and then loaded back later. Given a log recorded using the optimizations discussed in Section 3.3.4, the replay will proceed more efficiently.

3.4.4 Implementation

The replayer is implemented on TEMU, a dynamic analysis platform in the BitBlaze binary analysis infrastructure [15]. TEMU is based on QEMU version 0.9.1 and the changes described above were integrated into TEMU as well.

With the modifications to TEMU, existing analysis plugins should work automatically, except for a small change. Each regular TEMU plugin needs to check if the current execution is within the context of interest (e.g., if the current process is the malware's

process). However, for a plugin that works with replay, all execution is of interest. This is ensured by the recorder, so context checking in plugins is removed.

Two plugins were modified for experimentation. The first plugin is an unpacker, which is the implementation of Renovo [55]. The second is an instruction tracing tool called tracecap, which performs taint analysis and dumps detailed information for each instruction.

3.5 Evaluation

In order to assess the effectiveness of PHRR, V2E was evaluated in two ways. First, existing emulation detection techniques in the literature were studied and the effectiveness of V2E against these methods was examined. This test is focused on verifying transparency. Second, real-world malware samples were gathered, anti-emulation ones identified and subsequently studied using the two TEMU plugins. This latter test is focused on ensuring that transparent emulation based malware analysis is indeed feasible.

Experimental setup. The host machine has a Core i7 860 Quad Core processor with 4 GB of memory running Ubuntu 10.04 and kernel version 2.6.32.29 modified with the recorder logic. The guest systems are Windows XP SP2, Ubuntu 9.04 and Redhat 7.

3.5.1 Study of Existing Anti-emulation Attacks

To evaluate if V2E can defeat published anti-emulation methods, a list of such techniques was found in the literature [11, 12, 66] and then categorized based on how emulation differs from real hardware. Descriptions of these methods are listed in Table 3.2.

Description	Defeated?
`cpuid` returns processor specific information. QEMU returns generic information. [12]	✓
`rdtsc` returns the contents of the TSC, and can be used to measure elapsed time. [11, 12, 66].	✓
`cmpxch8b` conditionally writes to the memory operand, but a #GP exception is always generated if the memory operand is not writable. QEMU only raises #GP when the memory is written indeed. [12]	✓
A *double fault* exception is generated if the #GP handler can't be retrieved from the IDT when a #GP occurs. QEMU generates #GPs repeatedly. [12]	✓
Writing to *reserved MSRs* should generate #GP, but QEMU does not. [11]	✓
A #GP is generated if the *instruction length* is more than 15 bytes, but certain prefixes like `rep` can break this rule in QEMU. [11, 66]	✓
If the *alignment check* flag is set, then an alignment exception is thrown when an unaligned memory address is accessed. This exception is not thrown in QEMU. [11]	✓
CPU bugs or *errata* results in behavior that are either unexpected or does not follow the published logic, thus the emulator does not handle these like specific hardware would. [11]	✓∗
MSR fingerprinting uses the `rdmsr_safe` macro to find the MSRs supported by the CPU. The list of MSRs can be used to help determine the CPU model. [11]	✓
The *fnstcw* instruction pushes the FPU Control Word register onto the stack. As it turns out, bit 3 of this register is reserved in Intel's implementation but is always 1. QEMU always returns 0. [66].	✓
`icebp` is an *undocumented instruction* that simply raises an exception. In QEMU this instruction hangs the emulator. [66]	✓
`rep stosb` can be used to overwrite a range of memory with `nop`. When paired with a `jmp` to nowhere, QEMU and single stepping will throw a segmentation fault. Hardware executes successfully. [12]	N/A

Table 3.2: Survey of Emulation Detection Techniques.

Among these detection methods, six of them, including "cmpxch8b", "double fault", "reserved MSR", "instruction length", "alignment checking" and "invalid opcode", cause aberrant exception behavior. As V2E replays the exceptions from real hardware, these detection methods should be defeated. Defeating exception based emulation-detection was verified using synthetic sample programs of the *instruction length* and *invalid opcode* methods. The sample programs simply implement the methods as published. They were executed in QEMU, real hardware and V2E, and the expected exception behavior in V2E was verified manually.

"cpuid", "MSR fingerprinting" and "rdtsc" return deviated values. These values are hardware specific and thus hard to emulate correctly and completely. PHRR defeats these methods by recording the values the real hardware. These methods were also verified using synthetic samples and real-world malware samples.

"fnstcw" causes a deviated state in the FPU. V2E passes FPU/MMX/SIMD instructions directly onto hardware, so these deviations disappear in V2E. This was verified using several real-world samples.

The "CPU errata" method needs special consideration. If a CPU bug causes a totally unpredictable result, then it would be extremely hard to handle. However, the CPU bugs used for emulation detection in the literature [11] all cause incorrect exceptions. These CPU bugs can be handled correctly by PHRR, because exceptions are recorded and replayed. It is difficult to ascertain how prevalent non-exception based CPU bugs are. In general, the effectiveness of this detection method is limited, since CPU bugs are specific to a CPU family.

The "rep stosb" detection method exploits a cache coherency bug for self-modifying code in earlier Intel processors. This bug has been fixed in all current Intel processors. Therefore, this method is no longer relevant.

3.5.2 Analyzing Malware on Existing Malware Analysis Platforms

To determine how effective existing malware analysis platforms are at handling real-world malware, 150 real-world malware samples were collected from a live malware repository (`http://malc0de.com/database`) and security researchers. These samples were then tested on three malware analysis platforms: Anubis [126], CWSandbox [127], and TEMU [15]. While Anubis and TEMU are based on software emulation, CWSandbox uses API hooking.

Of the 150 samples, 51, 88, and 14 crashed or exhibited no behaviors in Anubis, CWSandbox, and TEMU respectively. Note that all these samples run properly in KVM, which means that they intended to escape from either of these analysis platforms. Interestingly enough, the 14 samples that are resistant to TEMU also evaded Anubis and CWSandbox. Evidently, anti-emulation malware has already become a prevalent threat.

3.5.3 Analyzing Real world Malware with V2E

To evaluate how well V2E handles real-world malware, the 14 anti-emulation samples were executed and recorded using V2E. A time-out threshold of 2 minutes was chosen to be consistent with the settings of Anubis and CWSandbox. For each sample, V2E was

			Null	Tracing		Unpacking	
MD5SUM	**exe sz**	**log sz**	**runtime**	**runtime**	**# ins**	**runtime**	**dumps**
27eb815f101a9295fbb601986f393d01	105KB	29.07MB	76.76s	2h19m	1347M	96.5s	78
43de1618764daf7e5887bd8ac9cadb52	105KB	28.46MB	76.69s	2h18m	1346M	96.09s	79
03f322365b844d8faf9236aab34b4214	106KB	30.75MB	77.09s	2h19m	1349M	97.02s	79
4f12dfb4b613abc4ddf56d087223a868	115KB	35.93MB	78.87s	2h20m	1366M	98.8s	57
f01cdf6e5052aeb5c6510bd8f8d88636	103KB	29.95MB	77.21s	2h19m	1348M	98.94s	81
f068b4362c646dae42cc3b1b8fe20c12	110KB	30.13MB	77.2s	2h19m	1350M	97.17s	77
1686739bc81a407dd9944e2d9bbcf2e1	23KB	2.44MB	0.65s	46.8s	7.73M	0.71s	8
0b8b2c0926630c69a6c75bba67b24a3e	39KB	3.25MB	0.97s	99.7s	16.8M	1.2s	55
c5ff7232868333107fa3efe895f12361	245KB	55.36MB	29s	27m15s	248M	39.33s	39
36e5fdcdbe0bcdc59ea001b162bfb97d	243KB	37.48MB	20.91s	1150s	175M	30.56s	22
c1a66699820fdeb7242e884e6d2f8bcb	119KB	676KB	0.57s	55.7s	9.55M	0.66s	10
dabec78d489f1e783fb23d6e726bd1a4	108KB	2.00MB	0.19s	23.2s	4.09M	0.22s	1
ef0458e196fbd1b4cc1613ba2ca3c43b	280KB	3.30MB	0.36s	54.8s	9.51M	0.46s	1
7ce6cd9837e1a7837c2b491c21ff5b69	101KB	7.10MB	34.35s	294.4s	43M	35.4s	30

Table 3.3: Analyzing Real-world Emulation-Resistant Malware with V2E

configured to record the entire user-level process and spawned child processes if any. Networking was also disabled to prevent malicious behavior from escaping the virtual machine sandbox.

V2E was able to record and replay the malicious behaviors of all these samples. In particular, three settings were used to test replay: 1) replay with no plugin provides a baseline for the replay performance; 2) replay with tracing produces a complete and detailed instruction trace for the recorded execution; and 3) replay with unpacking extracts hidden code and data from the packed malware. A summary of the results is presented in Table 3.3. For each sample, the MD5 hash, executable size, and size of the recorded execution log are listed first followed by the runtime for replay with no plugins. With regards to tracing, the instruction count and the runtime for tracing are listed in separate columns. As for unpacking, the number of memory dump files and the runtime for unpacking are shown.

The following observations can be made from Table 3.3. First, the execution logs (after compression) are fairly small (up to 55MB). It is worth noting that unlike the logs in the other execution replay systems, these logs are self contained with all necessary code and data included. They can be directly fed into the replayer for in-depth malware investigation and no other environmental setup (e.g., virtual machine images and configurations) is needed.

Second, due to the efficiency of dynamic binary translation, the baseline performance of the replayer (with no plugin) ranges from less than 1 second to 79 seconds. This is satisfactory since the malware sample was allowed to execute for 120 seconds (2 minutes) on real hardware. The very short replay runtime (less than 1 second) on some samples

indicates that these samples are mostly idle. This is reasonable because many of the samples are bots and networking is disabled during recording. Note that some samples are very computation-intensive with over 1.3 billion instructions executed within 2 minutes.

Third, the unpacker built on top of the replayer demonstrated good efficiency. It was able to finish replaying 2-minute execution logs in up to 99 seconds, and at the same time successfully extract hidden code and data from the packed malware samples. Interestingly enough, all of the samples are packed. Without V2E's support, it was not possible to unpack them successfully using TEMU. Finally, tracing is substantially more heavyweight than unpacking, because it has to disassemble each instruction, fetch instruction raw bytes and operands, and write these details into a file. The instruction traces completed within a reasonably short period (from tens of seconds to a couple of hours).

It is reasonable, because it is possible to configure the tracer to skip over certain instructions. To do so, the analyst can first replay a sample with the unpacker and determine the instruction at which unpacking finishes. Given that emulation detection methods are employed prior to unpacking (otherwise it will not serve its purpose of evading analysis), this can be considered the start of interesting behavior that should be further analyzed. The replayer can then be configured to only trace the instructions beyond that point. Applying this method to the six large samples resulted in a 99% reduction in the instruction trace size.

3.5.4 Recorder Performance

The malware evaluation provided some insights into the replayer's performance. Without ground knowledge about the real-world malware, it is difficult to accurately measure the performance of the recorder. Instead, the performance is estimated using controlled and synthetic experiments.

Recording adore-ng. The adore-ng rootkit is used to estimate recorder performance under frequent realm switches. The rootkit was installed into the Redhat 7 guest and exercised by decompressing the Linux kernel source with about 17,000 files. Since adore-ng intercepts file based system calls, there is at least one realm change per file. The workload took 3s without recording and 52s when recording was enabled, generating a 14MB execution log. A roughly 17x slowdown seems high, but is reasonable for this workload with frequent context switch between the rootkit and the rest of the kernel.

Recording Internet Explorer. Internet explorer was used to test how well V2E performs while recording a highly complex, computation-intensive, and interactive application. The load time of IE with and without recording was measured in this experiment. Without recording, IE started up and loaded the MSN homepage in 2.5s. With recording, it took 13.8s (about a 5x slowdown) and resulted in a 52MB execution log. The recording performance impact is expected to decrease as IE continues to run, because more pages (e.g., shared libraries) should remain stable in the recording realm. Despite this, a 5x slowdown is reasonable, since IE was still very responsive to user inputs while recording. It is worthwhile to note that the IE log did not replay successfully. This could be due to the

CPU based landmark or unknown emulation bugs, but the real reason remains unknown. Since the malware recorded and replayed correctly, fixing the error is left as future work.

Comparing with Single Stepping. Single stepping was implemented in KVM and a synthetic program that executes a loop with 8 million instructions were used to assess performance overhead. On KVM, it took approximately .008s vs. 25s when single stepping was disabled and enabled respectively. That is more than a 3000x slowdown. In contrast, the same 8 million instructions were recorded with negligible performance penalty (i.e., the recording time was approximately .008s). The baseline replay runtime was 0.3s, and it took only 0.8s to perform unpacking analysis and 48s to obtain the complete instruction trace.

3.6 Discussion

The limitations and potential evasion techniques are discussed in this section.

Bugs in Common Instructions. To achieve transparency, common instructions are assumed to emulate correctly. This assumption does not necessarily have to hold. If malware exploits a previously unknown emulation bug in common instructions, then replay will not be successful. Once the replay failure is found; however, the bug can be identified and its emulation code fixed.

Attacking the Landmarks. As mentioned in Section 3.3.7, the landmark mechanism is not perfect because the CPU state is not a unique identifier of an execution point. A malware author may take advantage of this limitation to force an imprecise replay. Resolving the issue by implementing a branch counter landmark is left as future work.

Multi-core Support. The current implementation of V2E only supports a single-core guest environment. Newer versions of QEMU supports multi-core guests and emulation based malware analysis platforms that support multi-core guests are expected in the future. Since TDP is used to separate the main and recording realms and each virtualized core has its own VMCS and TDP page table, recording malware execution on multi-core environments is feasible by design. Implementation is left as future work as well.

Denial-of-Service Attack. It is feasible for malware to induce a large number of exits (e.g., TDP page faults and exceptions) to the hypervisor, so as to launch a denial-of-service attack on the recorder. In addition, malware could detect the analysis environment by measuring this slowdown using an external clock. In general, this kind of limitation is not unique to V2E; it is also shared by other platforms (like Ether). Analysts will have to implement case-by-case solutions once they actually arise.

3.7 Conclusion

Precise heterogeneous record and replay and its implementation in V2E were presented in this chapter. In PHRR, a malware sample is allowed to execute under hardware virtualization where its actions are recorded for replay and further analysis. By analyzing and categorizing the expected differences between emulation and hardware virtualization, a recorder that not only captures all inputs, but also all deviant behaviors was designed. A corresponding dynamic binary translation based replayer that precisely replays the recorded events was also designed to ensure that, despite being emulated, the sample will execute exactly as it did under hardware virtualization.

In V2E, the recorder was implemented in KVM and the replayer in TEMU. Subsequently, a number of synthetic and real-world samples were used to determine whether PHRR as implemented in V2E is sufficient to help TEMU become transparent while maintaining its emulation based malware analysis advantages of efficiency and flexibility. V2E was successful in defeating previously published emulation detection techniques as well as 14 real-world malware samples that have previously evaded emulation based analysis. Furthermore, tests on recorder and replayer performance returned acceptable results. The recorder exhibited an approximately 17x performance degradation for a rootkit sample that caused many realm switches.

In summary, V2E showed that precise heterogeneous record and replay can help currently available emulation based dynamic binary analysis tools achieve *transparency.* The next focus of this dissertation is to ensure that emulation based dynamic binary analysis of mobile platforms is also feasible. Techniques for bridging the two semantic gaps for Android malware analysis and their implementation in DroidScope are presented in the next chapter.

4. EMULATION-BASED ANDROID MALWARE ANALYSIS

4.1 Introduction

Emulation based malware analysis has the advantages of *isolation* over user/kernel space based implementations, and *flexibility* and *efficiency* over hardware virtualization based ones. The major disadvantages are *transparency* and *semantic gaps*.

The previous chapter showed that precise heterogeneous record and replay is a viable technique to help emulators become more transparent, and therefore, the disadvantage due to transparency is reduced. Furthermore, virtual machine introspection techniques have proven to be effective in bridging the semantic gap between the emulator (or virtual machine monitor) and the guest operating system. This also lessens the disadvantage due to semantic gaps.

The semantic gap problem has resurfaced with the advent of mobile, and Android in particular, malware. There are two levels of semantic information that must be rebuilt. Android applications can contain native and Java components that cooperate in order to achieve a common goal; the Java components are interpreted by the Dalvik Virtual Machine, a Java Virtual Machine.

While traditional VMI techniques can successfully bridge the native level semantic gap (e.g., identify processes), new methods are needed to reconstruct the Java level semantic information (e.g., which Java method is being executed). Ideally, the two levels of semantic

information are also bound together so that the native and Java components' execution and interactions between them can be analyzed using one single tool. The details on how this is achieved are presented in this chapter.

The general architecture of a new emulation-based Android malware analysis platform is described in Section 4.2. This architecture is implemented in a tool named DroidScope. The architecture includes an Instrumentation Interface that analysts can use for plugin development. The details on how the interface maintains efficiency and flexibility are presented in Section 4.3. The architecture also includes two virtual machine introspection libraries, one for the native context and one for the Java context. The details on the different data structures needed for VMI are presented in Sections 4.4.1 and 4.4.2 for the native and Java contexts respectively. The techniques are implemented in DroidScope and several plugins were also implemented to evaluate the effectiveness of this new analysis platform. These plugins are also evaluated in terms of performance and ability to analyze real-world Android malware with cooperating Java and native components. The plugins and results are presented in Sections 4.5 and 4.6. Limitations are discussed in Section 4.7 and an intermediary conclusion drawn in Section 4.8.

4.2 Architecture

Figure 4.1 depicts the architecture of an emulation based analysis platform for Android malware. The design is implemented as a tool named DroidScope and thus for brevity, DroidScope will be used to refer to both the architecture and the implementation.

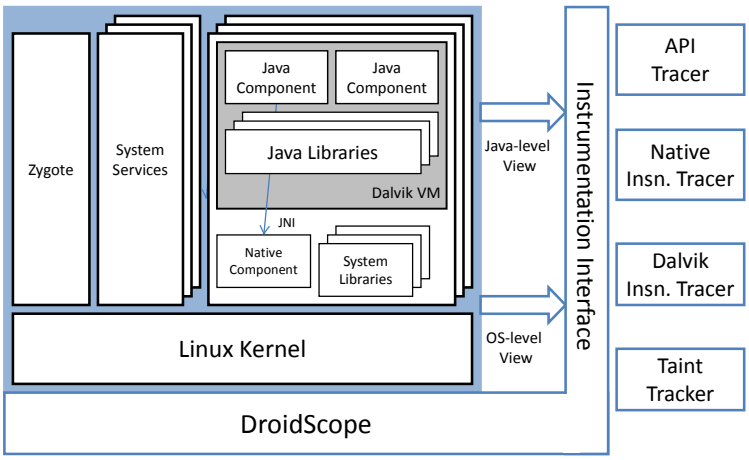

Fig. 4.1.: DroidScope Overview

Like other emulation based malware analysis architectures, the entire Android system (including the malware) runs on top of an emulator - the Android emulator in this case -; the analysis is completely performed from the outside. In this case, DroidScope is built on top of the QEMU based Android emulator that ships with the Android SDK (Software Development Kit) to ensure the best compatibility with virtual Android devices. There are important components to the architecture: native-level view, Java-level view and the Instrumentation Interface (II).

The native-level view includes a machine-level view that exposes low level information such as instructions and raw memory access to the analyst and an OS-level view that rebuilds OS constructs such as processes and system calls. The Java-level view uses VMI to interpret the internal state of Java components including Java objects in memory and the state of the Dalvik Virtual Machine (a Java Virtual Machine). The Instrumentation Interface abstracts away the details of how intrumentation (i.e., the technique used to execute analysis code alongside guest code) takes place so the analyst can focus on what to

do with instrumentation (e.g., implement an instruction tracer). The interface also includes access to a library of VMI related functions so the analyst can readily access the two reconstructed views.

To complete the overall emulation-based analysis platform, the figure also depicts a number of plugins that can be built on this new architecture. An *API tracer* can be implemented to monitor the malware's activities at the API level. This can then be used to reason about how the malware interacts with the Android runtime environment. Since Android envrionment includes both the Java framework as well a native libraries, the API tracer should not only monitor how the malware's Java components communicate with the Android Java framework and how the native components interact with the Linux system but also how Java components and native components communicate through the JNI interface. This is possible since the native and Java level views are available to the API plugin at all times. This plugin can be used to illustrate VMI as well as the flexibility aspect of emulation based malware analysis.

The *native instruction tracer* and *Dalvik instruction tracer* are plugins for looking into how a malicious App behaves internally by recording detailed instruction traces. The Dalvik instruction tracer records Dalvik bytecode instructions for the malware's Java components and the native instruction tracer records machine-level instructions for the native components (if they exist). The instruction tracers can be used to illustrate how efficient emulation based analysis is.

Taint analysis is one of the other fine-grained analysis techniques that are suitable for implementation in an emulation based analysis platform. The *taint tracker* plugin is an implementation of this technique. Since taint analysis is an important dynamic analysis

technique, the core taint propagation logic is implemented as an internal-plugin that is part of the Instrumentation Interface. The taint tracker plugin is simply a wrapper. This means that other plugins, such as the Dalvik Instruction Tracer can also use taint analysis to enrich the traces. It is worth noting that, by design, dynamic taint analysis is implemented at the machine code level only. It is assumed that with semantic knowledge at both native and Java levels, information leakage in Java components, native components, or even collusive Java and native components and be revealed. This assertion is verified to be true for arithmetic operations in the next Chapter.

4.3 Instrumentation Interface

The Instrumentation Interface serves as the interface between plugins and DroidScope's internal logic and the analysis plugins. It serves two main purposes, export the virtual machine introspection functions and methods to the plugins and allow plugins to easily instrument or intercept the malware sample's execution. Virtual machine introspection requires instrumentation support so important points of a guest's execution can be detected and analysis can be conducted. The changes made to the Android emulator for instrumentation is discussed next.

4.3.1 Basic Instrumentation

Recent versions of QEMU, like the one the Android emulator is based on, use the Tiny Code Generator (TCG) to compile guest code blocks into host code blocks. The execution flow is as follows: 1) a basic block of guest instructions is disassembled and translated into

an intermediate representation called TCG-IR; 2) the TCG-IR code block is then compiled down to a block of host instructions and stored in a code cache; and 3) control jumps into the translated code block and guest execution begins. Subsequent execution of the same guest basic blocks will skip the translation phase and directly jump into the translated code block in the cache.

The crux of code instrumentation lies in the technique used to ensure that the program analysis code executes alongside the guest's code. This implies that there must be a way insert program analysis code into the translated code blocks used in step 3 above. To do so, extra TCG-IR instructions are inserted during the code translation phase (step 1), such that this extra analysis code is executed in the execution phase (step 3). For example. in order to monitor context switches, several TCG-IR instructions are inserted to call a helper function whenever the translation table registers (ARM system control co-processor c2_base0 and c2_base1 in QEMU) are written to. The logic for identifying the switched-out process or switched-in process can then be implemented in the helper function.

Instrumentation Callbacks The problem with directly inserting analysis code into helper functions is it limits the flexibility of analysis plugins. This can be better illustrated using instruction level instrumentation where a helper function is called whenever a guest instruction is executed. It is conceivable that two different analysis plugins will require instruction level instrumentation. For example, an instruction tracer plugin will have logic that disassembles each instruction and writes the contents to a log. Whereas, a control flow graph generator plugin will have logic that checks whether the instruction is a branch instruction and if so, add an extra node and/or edge to the control flow graph. In the basic

scheme, the logic for both plugins will be implemented in the same helper function coupling their functionalities. A bug in one plugin will affect the other.

A better solution is to abstract away the details of adding the instrumentation code and export an event based callback interface. In this design, the plugin logic is implemented in separate and distinct functions. Each plugin can then register a callback such that their respective functions are called whenever an instruction is executed. The plugins can also unregister the callbacks when their jobs are complete. Thus, the purpose of the helper function is to search through the registered callback functions and issue the appropriate calls. This is a role of the Instrumentation Interface.

4.3.2 Application Programming Interfaces

Abstracting away the details of inserting instrumentation code is only one function of the II. The other functions are to expose the native and Java level views to analysis plugins. All of the different functions can be categorized into three different Application Programming Interfaces (APIs) to mirror the different context levels of an Android device: the native or machine API, the OS or Linux API and the Java or Dalvik API. These APIs can also be further separated into two sub-categories. Analysts can register event based callbacks using the *Events* sub-API so they are notified of when certain events of interest take place. They can then use the *Query and Set* sub-APIs to interpret and potentially change the guest's state using VMI. Table 4.1 summarizes these APIs. The details are presented in Sections 4.4.1 and 4.4.2. A short description is provided below.

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

Table 4.1: Summary of DroidScope APIs

At the native level, one can register callbacks for instruction start and end, basic block start and end, memory read and write, and register read and write. One can also read and write memory and register content. As taint analysis is implemented at the machine code level, one can also set and check taint in memory and registers.

At the OS level, one can register callbacks for context switch, system call, task start, update (such as process name), and end, and memory map update. One can also query symbols, obtain the task list, and get the current execution context (e.g., current process and thread).

At the Dalvik level, one can instrument at the granularity of Dalvik instructions and methods. One can query the Dalvik symbols, parse and interpret Java objects, read and modify DVM state, and selectively disable the Just-In-Time (JIT) compiler in the DVM for certain memory regions. Through the Dalvik-view, one can also set and check taint in Java Objects as well.

4.3.3 Instrumentation Optimization

A general guideline for performance optimization in dynamic binary translation is to shift computation from the execution phase to the translation phase. For instance, if the analyst one needs to instrument a function call at address x using basic blocks, then one should insert the instrumentation code for the block at x when it is being translated instead of instrumenting every basic block and look for x at execution time.

This guideline is followed as part of the new analysis platform. Consequently, the instrumentation logic becomes more complex. When registering for an event callback, one can specify a specific location (such as a function entry) or a memory range (to trace instructions or functions within a particular module). Therefore, the instrumentation logic and the APIs support single value comparisons and range checks for controlling when and where event callbacks are inserted during the translation phase.

The instrumentation logic is also dynamic, because analysts can register and unregister a callback at execution time. For example, when the virtual device starts, only the OS-view instrumentation is enabled so the Android system can start quickly as usual. When the analyst starts analyzing an App, instrumentation code is inserted to reconstruct the Dalvik view and to perform analysis as requested by the plugin. When instrumenting a function return, the return address will be captured from the ARM link register, `r14`, at the function entry during execution, and a callback registered at the return address. After the function returns, this callback is removed since it has served its purpose. Then when the analysis has finished, other instrumentation code is removed as well.

In order to support dynamic instrumentation, QEMU's translated code cache must be flushed whenenver a new callback is registered or un-registered. However, flushing the whole cache means that all previously translated code blocks, even ones that are unaffected by the changes in callback registration, will need to be translated again. This introduces unnecessary performance overhead. Thus, by design, the instrumentation logic should only flush (i.e., invalidate) the translated blocks that are affected by the change. For example, when the analyst removes the callback for the function return, only the translated block that starts at the return address and blocks that point to it are invalidated. The rest of the cache is left intact.

4.3.4 Taint Analysis

Taint analysis is an important dynamic analysis technique and has been implemented as part of many different analysis platforms. Therefore, support for taint analysis is included as part of the Instrumentation Interface. A simple taint analysis design is used. In this design, each byte of data is labeled either as tainted or untainted and taint propagates through native instructions only. It is assumed that since native instructions are used emulate Dalvik bytecode, tracking taint through native instructions should be sufficient for tracking taint trough the bytecode. This assumption needs to be verified. Furthermore, the design is simple since the taint propagation policy uses a simple "or" rule. That is, the result of an operation is tainted as long as any of the operands are tainted.

While setting and checking the taint of native objects is straight forward, setting and checking Java objects involves some understanding of how Java objects are represented in

memory (see Section 4.4.2). Briefly, tainting an object involves first separating the real data (e.g., an objects fields) from the metadata (e.g., a field's name) and then labeling the real data as tainted. Checking taint is done in a similar manner.

4.4 Bridging the Semantic Gaps

This section discusses the methodology for rebuilding the two levels of semantic views. Details for rebuilding information about processes, threads, memory mappings and system calls at runtime are described first (the OS-level view) followed by details about the Dalvik Virtual Machine and rebuilding the Java or Dalvik-level view.

4.4.1 Reconstructing the OS-level View

The native-level view is essential for analyzing native components. The machine-level view provides insight into low-level execution details such as the native instructions being executed. Since the details are defined as part of the Application Binary Interface and Instruction Set Architecture documentation, it will not be elaborated further here. The focus is on the OS-level view.

The basic techniques for reconstructing the OS-level view have been well studied for the x86 architecture and are generally known as virtual machine introspection [17–19]. The core idea is to understand which data structures contain pertinent information and how the data structures can be reached and interpreted from the low level, raw, view of the virtual machine's state. This section focuses on the kind of information that is made available to analysts through the II either in the form of new events or query and set functions.

System Calls A user-level process has to make system calls to access various system resources and thus obtaining its system call behavior is essential for understanding malicious Apps. On the ARM architecture, the service zero instruction `svc #0` (also known as `swi #0`) is used to make system calls with the system call number in register `r7`. This is similar to x86 where the `int 0x80` instruction is used to transition into privileged mode and the system call number is passed through the `eax` register.

To obtain system call information, additional TCG-IR instructions are inserted to call a helper function whenever the special instructions above are translated (i.e., the special instructions are instrumented). This helper function then dispatches the system call event to functions that registered for the even using the II.

For example, a plugin that logs system calls can simply register for the event and when notified, log the program counter of the caller and the system call number. For important system calls (e.g., open, close, read, write and connect), the system call parameters and return values can also be retrieved as well. The parameters are read them from the general purpose registers and/or the stack and interpreted based on the published system call interface. Block begin events are registered for the return address of the system call and the return value is retrieved when the event callback function is notified.

Shadow Task List From the operating system perspective, Android Apps are user-level processes. Therefore, it is important to know what processes are active and which one is currently running. A shadow task list with select information about each task is maintained to make this information readily available to analysis tools. The shadow task

list can be accessed through the II. Analysts can also register for events based on whether a new task has been added, removed or the details have changed.

The basic executable unit in the Linux kernel is the task, which is represented by the `task_struct` structure. A list of active tasks is maintained in a `task_struct` list which is pointed to by `init_task`. To distinguish between a thread and a process, a task's process identifier `pid` as well as its thread group identifier `tgid` is retrieved from the guest's memory space. The `pgd` (the page global directory that specifies the memory space of a process), `uid` (the unique user ID associated with each App), and the process' name are also maintained as part of the shadow task list. Additionally, experience has shown that malware often escalates its privileges or spawns child processes to perform additional duties. Thus, the shadow task list also contains the task's credentials (i.e., `uid, gid, euid, egid` as well as the process' parent `pid`).

Special attention is paid to a task's name since the `comm` field in `task_struct` can only store up to 15 characters. This is often insufficient to store the App's full name, making it difficult to pinpoint a specific App. This is also a simple example of how malware analysis in desktops differs from mobile systems. To address this issue, the complete application name is obtained from the command line string `cmdline`, which is pointed to by the `mm_struct` structure pointed to by `task_struct`. Note that the command line string is located in user-space memory, which is not shared like kernel-space memory where all the other structures and fields reside. To put it differently, the guest virtual addresses in all other structures and fields can be translated into guest physical addresses using the page tables of any guest process. In contrast, the GVA for `cmdline` can only be translated into the correct GPA using that process' page table.

According to the design of the Linux kernel, the `task_struct` for the current process can be easily located. The current `thread_info` structure is always located at the (*stack pointer & 0x1fff*), and `thread_info` has a pointer pointing to the current `task_struct`. All active tasks are identified by iterating through the doubly linked `task_struct` list. To ensure that the shadow list is up to date, it is updated whenever the *sys_fork*, *sys_execve*, *sys_clone* and *sys_prctl* system calls are executed.

Process Logger. Experience has shown that a history of all the threads and processes in a system is useful for quickly understanding the parent and child relationships between the different processes over time. Thus, a plugin that registers for the task begin and end events and logs the detailed information about the process is implemented as part of the OS-level view.

Memory Map A process' memory map reveals how a process' virtual memory space is segmented and the intended purpose of each segment. This information is useful for analysts and is needed for reconstructing the Java-level view. This is especially true for newer versions of Android, such as Ice Cream Sandwich, with address space layout randomization enabled by default. The virtual address of a library can be directly retrieved from the memory map, even if the virtual address changes with each execution of a program.

Similar to the shadow task list, a shadow memory map is built and made available to analysts as part of the OS-level view. Plugins can also register for the memory map updated event.

The shadow memory map is buit by iterating through the list of virtual memory areas by following the `mmap` pointer in the `mm_struct` pointed to by the `task_struct`. The information gathered includes the address range, the permissions and the name of the file if it is mapped. Also like the shadow task list, the shadow memory map information is updated when a system call (*sys_mmap2*) returns. Note that each process has its own shadow memory map. Due to this, only the memory map for the currently executing process is updated.

4.4.2 Reconstructing the Dalvik View

Reconstructing the Java or Dalvik view requires knowledge of how the DVM operates as well as the shadow lists in the OS-level view. The goal of the Dalvik view is to allow analysts to interpret Dalvik instructions, the current DVM machine state and even Java Objects. The pertinent details are presented in this section. Note that the following descriptions are primarily based on the ARM architecture. Some details on the x86 architecture are provided as needed. Furthermore, the descriptions are for Android Gingerbread. While the concepts should remain the same, some details might differ from version to version. This in turn means that the introspection implementations will need to updated to match the changes in the future.

Dalvik Instructions and mterp The DVM's main task is to execute Dalvik bytecode instructions. In Gingerbread and thereafter, it does so in two ways: interpretation and Just-In-Time compilation (JIT) [128].

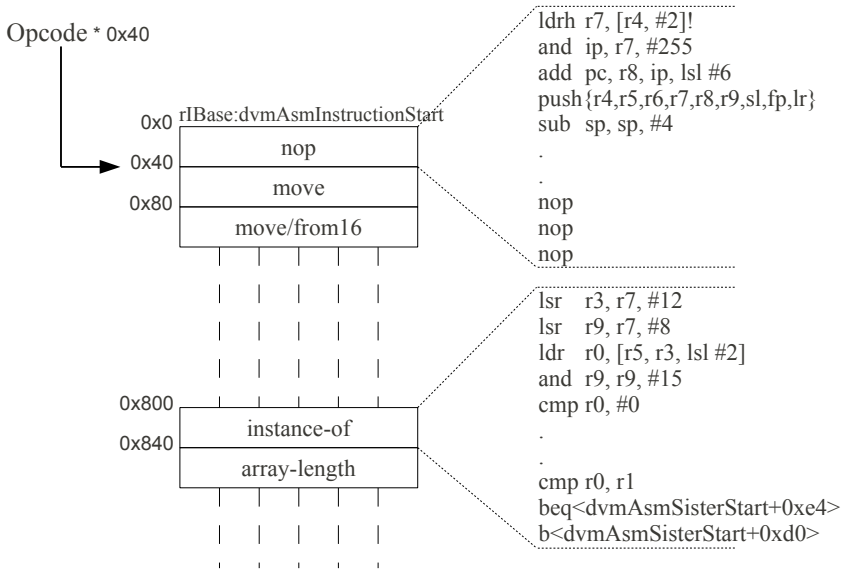

Fig. 4.2.: Dalvik Opcode Emulation Layout in `mterp`

The interpreter, named *mterp*, uses an offset-addressing method to map Dalvik opcodes to machine code blocks as shown in Figure 4.2. Each opcode has 64 bytes of memory to store the corresponding emulation code, and any emulation code that does not fit within the 64 bytes use an overflow area, `dvmAsmSisterStart`, (see `instance-of` in Figure 4.2). This design simplifies the emulation of Dalvik instructions. mterp simply calculates the offset ($opcode * 64$) and jumps to the corresponding emulation block.

This design also simplifies the reverse conversion from native to Dalvik instructions as well: when the program counter (`R15`) points to any of these code regions, the DVM is interpreting a Dalvik bytecode instruction. Furthermore, it is trivial to determine the opcode of the currently executing Dalvik instruction. The formula is $(R15 - rIBase)/64$, where rIBase is the virtual address of the beginning of the emulation code region. rIBase is

dynamically calculated as the virtual address of *libdvm.so* (obtained from the shadow memory map in the OS-level view) plus the offset of `dvmAsmInstructionStart` (a debug symbol). The emulation code for Dalvik opcode number `0` (`nop`) can be used as a signature to search for the start of the emulation section if the debug symbol is not available.

The Dalvik view registers for the block begin events in order to determine when a Dalvik bytecode instruction begins and issue the corresponding Dalvik Instruction Begin event that is part of the II. Since there are only 256 possible opcodes and each opcode takes only 64 bytes, only the block begin events for the pages that contain the emulation code are registered for. In other words, the instrumentation logic described previously will not insert the TCG-IR to call the helper function for any other basic blocks except the ones that contains Dalvik emulation code or were requested by other plugins. Furthermore, the Dalvik view only registers for these events if there are plugins that registed for the Dalvik instruction begin event. Once again, the event based callback interface is very dynamic.

Detecting the beginning of a method involves keeping track of whether the previous Dalvik instruction was one of the `invoke*` instructions along with making consistency checks (e.g., making sure it is in the same thread). Optimized block begin callbacks are also used for this purpose.

Selectively Disabling JIT The Just-In-Time compiler was introduced to improve performance by compiling heavily used, or "hot", Dalvik instruction traces (consisting of multiple code blocks) directly into native machine code. While each translated trace has a single entry point, there can be multiple exits known as *chaining cells*. These chaining cells either chain to other translated traces or to default entry points of the mterp interpreter.

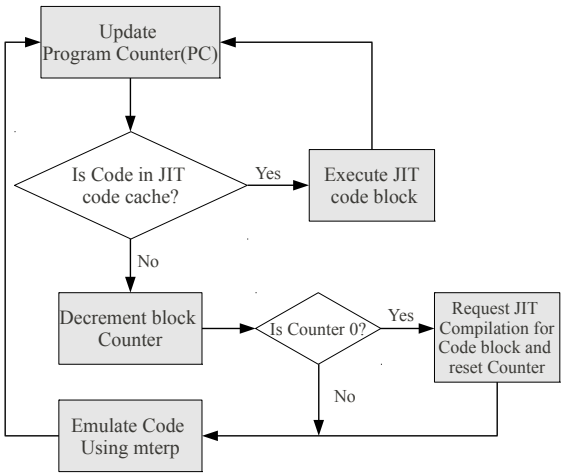

Fig. 4.3.: High Level Flowchart of *mterp* and JIT

Overall, JIT provides an excellent performance boost for programs that contain many hot code regions, although it makes fine-grained instrumentation more difficult. This is because JIT performs optimization on one or more Dalvik code blocks and thus blurs the Dalvik instruction boundaries.

An easy solution would be to completely disable JIT at build time, but it could incur a heavy performance penalty and more importantly it requires changes to the virtual device, which can lead to transparency problems. Considering that analysts are often only interested in the behavior of a particular section of Dalvik bytecode (such as the main program but not the rest of system libraries), an alternative solution is to *selectively* disable JIT at runtime. Analysis plugins specify the code regions for which to disable JIT using the II and as a result only the Dalvik blocks in those regions incur the performance penalty. All other regions and Apps still benefit from JIT.

Figure 4.3 shows the general flow of the DVM. It is similar to the dynamic binary translation steps taken by QEMU. When a basic block of Dalvik bytecode needs to be

emulated, the Dalvik program counter is updated to reflect the new block's address. That address is then checked against the translation cache to determine if a translated trace for the block already exists. If it does, the trace is executed. If it does not then the profiler will decrement a counter for that block. When this counter reaches 0, the block is considered hot and a JIT compilation requested. Compilation takes place in another thread. To prevent thrashing, the counter is reset to a higher value and emulation using mterp commences. As can be seen in the flow chart, as long as the requested code is not in the code cache, then mterp will be used to emulate the code.

The `dvmGetCodeAddr` function is used to determine whether a translated trace exists. It returns `NULL` if a trace does not exist and the address of the corresponding trace if it does. Thus, to selectively disable JIT, the DVM is instrumented to set the return value of `dvmGetCodeAddr` to `NULL` for any translated trace that needs to be disabled. In this case, instrumentation involves registering a callback for when the function returns, and when it does, alter the return value to `NULL`. This process is the same as the one used to update the shadow lists in the OS-level view, except for the fact that the return value is changed in this case.

It is imperative that changing the return value does not change the program's original flow. The following arguments are used to show that this is indeed true. They are based on the source code for Android Gingerbread and so the validity of these arguments might have to be revisited for other Android versions if the JIT logic changed.

First, if the original return value was `NULL` then the change will not have any side effects. Second, if the return value was a valid address, then by setting it to `NULL`, the profile counter is decremented and if 0 (i.e., the code region deemed hot again) another

compilation request is issued for the block. In this case, the code will be recompiled taking up space in the code cache. This can be prevented by not instrumenting the `dvmGetCodeAddr` call from the compiler. In addition to preventing the translated trace from being executed, setting the value to `NULL` also prevents it from being chained to other traces. This is the desired behavior.

For the special case where a translated trace has already been chained and thus `dvmGetCodeAddr` is not called, the JIT cache should be flushed whenever the disabled JIT configuration changes (e.g., JIT for a new code region needs to be disabled). This can be done by marking the JIT cache as full during the next garbage collection event, which leads to a cache flush. Once again these changes are made from the VMM through instrumentation. While this is not a perfect solution, it has been sufficient for the evaluations. Full JIT support is left as future work.

In all cases, the only side effect is wasted CPU cycles due to compilation; the execution logic is unaffected. Therefore, the side effects are inconsequential.

DVM State Figure 4.4 illustrates how the DVM maintains the virtual machine state. When mterp is emulating Dalvik instructions, the ARM registers `r4` through `r8` store the current DVM execution context. More specifically, `r4` is the Dalvik program counter, pointing to the current Dalvik instruction. `r5` is the Dalvik stack frame pointer, pointing to the beginning of the current stack frame. `r6` points to the `InterpState` data structure, called `glue`. `r7` contains the first two bytes of the current Dalvik instruction, including the opcode. Finally `r8` stores the base address of the mterp emulation code for the current DVM instruction. In x86, `edx`, `esi`, `edi` and `ebx` are used to store the program counter,

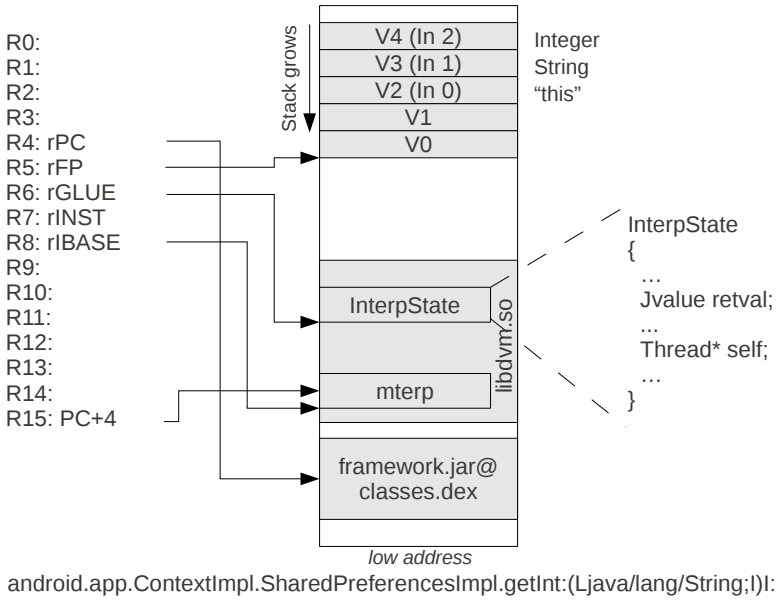

Fig. 4.4.: Dalvik Virtual Machine State

frame pointer, mterp base address and the first two bytes of the instruction respectively. The `glue` object can be found on the stack at a predefined offset. Dalvik virtual registers are 32 bits and are stored in reverse order on the stack. They are referenced relative to the frame pointer `r5`. Hence, the virtual register `v0` is located at the top of the stack (pointed to by the ARM register `r5`,) and the virtual register `v1` sits on top of `v0` in memory, and so forth. All other Dalvik state information (such as return value and thread information) is obtained through the `glue` data structure pointed to by `r6`.

By understanding how the DVM state is represented in the CPU registers and memory, detailed information such as the current DVM program counter, frame pointer and all virtual registers can be retrieved at runtime. The only requirement is knowing when the mterp interpreter is executing. In the minimum, this is true whenever the instruction pointer is pointing to the Dalvik bytecode emulation section.

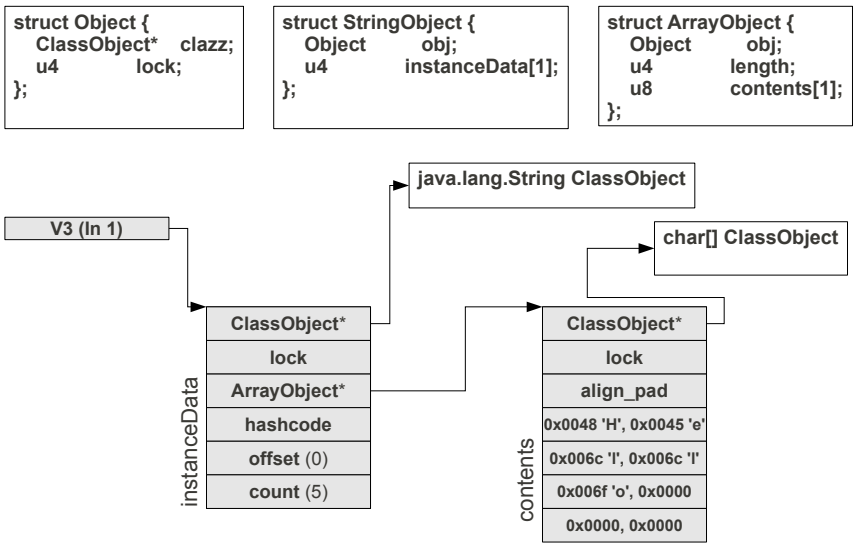

Fig. 4.5.: String Object Example

Java Objects Java Objects are described using two data structures. Firstly, *ClassObject* describes a class type and contains important information about that class: the class name, where it is defined in a dex file, the size of the object, the methods, and the location of the member fields within the object instances. To standardize class representations, Dalvik creates a ClassObject for each defined class type and implicit class type (e.g., arrays). For example, there is a ClassObject that describes a `char[]` which is used by `java.lang.String`. Moreover, if the App has a two dimensional array (e.g., `String[][]`), then Dalvik creates a ClassObject to describe the `String[]` and another to describe the array of the previously described `String[]` class.

Secondly, as an abstract type, *Object* describes a runtime object instance (i.e., the member fields). Each Object has a pointer to the ClassObject that it is an instance of plus a *tail accumulator array* for storing all member fields. Dalvik defines three types of Objects,

DataObject, *StringObject* and *ArrayObject* that are all pointed to by generic `Object*`s. The correct interpretation of any `Object*` fully depends on the ClassObject that it points to.

A simple String ("Hello") is used to illustrate the interpretation process. Figure 4.5 depicts the different data structures involved as well as the struct definitions on top. The String is referenced by the virtual register `v3`. Since Java references are simply `Object*`s, `v3` points to an Object. To determine the type of the object, the first 4 bytes of the object is used to reach the ClassObject structure. This ClassObject instance describes the `java.lang.String` class. Internally, Dalvik does not store the String data inside the StringObject and instead use a `char[]`. Consequently, `instanceData[0]` is used to store the reference to the corresponding `char[]` object and `instanceData[3]` is used to store the number of characters in the String, 5 in this case.

Then, the String's data is obtained by following `instanceData[0]` to the character array. Once again the `Object*` within the new object must be used to correctly interpret it as an ArrayObject. Note that since ARM EABI (Embedded Application Binary Interface) requires all arrays to be aligned to its element size and `u8` is 8 bytes in length, an implicit 4 byte `align_pad` was inserted into the ArrayObject to ensure that the `contents` array is properly aligned. Given the length of the String from the StringObject and the corroborating length in the ArrayObject, the "Hello" String is found in the `contents` array encoded in UTF-16.

4.4.3 Symbol Information

Symbols (such as function name, class name, field name, etc.) provide valuable information for human analysts to understand program execution. Thus, a symbol database is maintained for access through the II. For portability and ASLR support, a single database of offsets to symbols is used per module. At runtime, finding a symbol by a virtual address requires first identifying the containing module using the shadow memory map, and then calculating the offset to search the database.

Native library symbols are retrieved statically through *objdump* and are usually limited to Android libraries since malware libraries are often stripped of all symbol information. On the other hand, Dalvik or Java symbols are retrieved dynamically and static symbol information through *dexdump* is used as a fallback. This has the advantage of ensuring the best symbol coverage for optimized dex files and even dynamically generated Dalvik bytecode.

More DVM data structures are used to retrieve symbols at runtime. For example, the *Method* structure contains two pointers of interest. `insns` points to the symbol address (in other words, the start of the method's bytecode) and `name` points to the method's name (this field is located in the memory mapped dex file). Conveniently, the `glue` structure pointed to by `R6` has a field, `method`, that points to the Method structure for the currently executing method.

There are times when this procedure fails though, e.g., if the corresponding page of the dex file has not been loaded into memory yet. In these cases, the offset into the dex file can be calculated using the shadow memory map and the information retrieved from a local

copy of the corresponding dex file. If this fails as well, then the static symbol information from *dexdump* is used as a last resort. This same basic method of relying on the DVM's data structures is used to dynamically retrieve class and field names as well.

4.5 Plugins

The architecture including the Instrumentation Interface, the OS-level view and Java or Dalvik-level views were implemented in a tool named DroidScope. DroidScope is built on top of the Android emulator that ships with the Android Gingerbread source. The Android emulator is in turn based on QEMU version 0.10.50. This section discusses the plugins that were implemented to illustrate the flexibility of analysis tool development.

4.5.1 Sample Plugin

Figure 4.6 presents sample code for implementing a simple Dalvik instruction tracer. This is not a real plugin and the functions prototypes are not the ones found in the actual DroidScope implementation. The *_init* function at L19 will be called once this plugin is loaded in DroidScope. The *_init* function specifies which program to analyze by calling the *setTargetByName* function. It also registers a callback *module_callback* to be called when module information is updated. *module_callback* will check if the DVM is loaded and if so, disable JIT for the entire memory space (L9 and L11.) It also registers a callback, *opcode_callback*, for Dalvik instructions. When called, *opcode_callback* prints the opcode information.

```
 1. void opcode_callback(uint32_t opcode) {
 2.   printf("[%x] %s\n", GET_RPC, opcodeToStr(opcode));
 3. }
 4.
 5. void module_callback(int pid) {
 6.   if (bInitialized || (getIBase(pid) == 0))
 7.     return;
 8.
 9.   gva_t startAddr = 0, endAddr = 0xFFFFFFFF;
10.
11.   addDisableJITRange(pid, startAddr, endAddr);
12.   disableJITInit(getGetCodeAddrAddress(pid));
13.   addMterpOpcodesRange(pid, startAddr, endAddr);
14.   dalvikMterpInit(getIBase(pid));
15.   registerDalvikInsnBeginCb(&opcode_callback);
16.   bInitialized = 1;
17. }
18.
19. void _init() {
20.   setTargetByName("com.andhuhu.fengyinchuanshuo");
21.   registerTargetModulesUpdatedCb(&module_callback);
22. }
```

Fig. 4.6.: Sample code for Dalvik Instruction Tracer

This sample code will print all Dalvik instructions for the specified App, including the main program and all libraries. If the analyst is only interested in the execution of the main program, he or she can add call the II function *getModAddr("example@classes.dex", &startAddr, &endAddr)* at L10. This function locates the dex file in the shadow memory map and stores its start and end addresses in the appropriate variables. The rest of the code can be left untouched.

4.5.2 Analysis Plugins

To demonstrate the flexibility of analysis plugin development as well as DroidScope's ability to analyze Android malware, four analysis plugins have been implemented: API tracer, native instruction tracer, Dalvik instruction tracer, and taint tracker.

API tracer monitors how an App (including Java and native components) interacts with the rest of the system through system and library calls. First all of the App's system calls are logged by registering for system call events. A whitelist of the virtual device's built-in native and Java libraries is then created to list all of the libraries that should not be traced. As modules are loaded into memory, any library not in the whitelist is marked for analysis. The *invoke** and *execute** Dalvik bytecodes are used to identify and log method invocations, including those of the sample. Since this is a small number of instructions, block begin events at the start of the instructions' emulation code are used to determine when these Dalvik bytecode instructions are being emulated instead of the Dalvik instruction begin events. This improves performance.

The log contains the currently executing Java thread, the calling address, the method being invoked as well as a dump of its input parameters. Since Java Strings are heavily used, all Strings are converted into native strings before logging when possible. Then, the *move-result** bytecode instructions are instrumented to detect when system methods return and gather the return values. Once again, block begin events are used in lieu of Dalvik instruction begin events.

To log library calls from the App's native components, block end events for blocks that are located in the App's native components are used. When the callback for a block end event is invoked, the address of the next block is check to see whether it is within the App's native components. If not, then it signals a control flow transition from the App to the system libraries, and so the event is logged.

Native instruction tracer registers ARM or x86 instruction callbacks to gather information about each instruction including the raw instruction, its operands (register and memory) and their values. These are logged into files.

Dalvik instruction tracer follows the basic logic of the above example and logs the decoded instruction to a file in the *dexdump* format. The operands, their values and all available symbol information (e.g., class, field and method names), are logged as well.

Taint tracker utilizes the dynamic taint analysis APIs to analyze information leakage in an Android App. It specifies sensitive information sources (such as IMEI, IMSI, and contact information) as tainted and keeps track of taint propagation at the machine code level until they reach sinks, e.g. *sys_write* and *sys_send*.

With the OS and Dalvik views, it further creates a graphical representation to visualize how sensitive information has leaked out. Whenever taint is propagated, a node that represents the currently executing function or method and nodes for the tainted memory locations are added. Since methods operate on Java Objects, an attempt is made to identify the containing Object and a node created for it instead of the simple memory location. This is done for a method's input parameters, the current object (i.e., "this") and returned objects.

4.6 Evaluation

The simplicity of the event based instrumentation interface and the analysis plugins that were built using the Instrumentation Interface showed that DroidScope, like other emulation based malware analysis platforms, is flexible for analysis plugin development.

What remains to be show is that the infrastructure and plugins do not impose too much of a performance overhead (i.e., is efficient) and that the plugins are capable of analyzing real world Android malware with both Java and native components.

Seven benchmark Apps from the official Android Market are used to evaluate introspection and plugin performance. The Apps are: AnTuTu Benchmark (ABenchMark) by AnTuTu, CaffeineMark by Ravi Reddy, CF-Bench by Chainfire, Mobile processor benchmark (Multicore) by Andrei Karpushonak, Benchmark by Softweg, and Linpack by GreeneComputing. The results are presented in Section 4.6.1.

Two real world Android malware samples, DroidKungFu and DroidDream, are analyzed using the developed plugins to evaluate capability. The detailed analysis results are presented in Sections 4.6.2 and 4.6.3 respectively.

Experimental Setup All experiments were conducted on an Acer 4830TG with a Core i5 @ 2.40GHz and 3GB of RAM running Xubuntu 11.10. The Android guest is a Gingerbread build configured as "user-eng" for ARM with the Linux 2.6.29 kernel and uses the QEMU default memory size of 96 MB. No changes were made to the Android source unless otherwise noted.

4.6.1 Performance

To measure the performance impact of instrumentation, the analysis plugins were used to analyze the various benchmark Apps while the Apps performed their tests. This was repeated 5 times. The default Android emulator without any instrumentation is used as a baseline. Also, since DroidScope selectively disables JIT on the Apps, the Android source

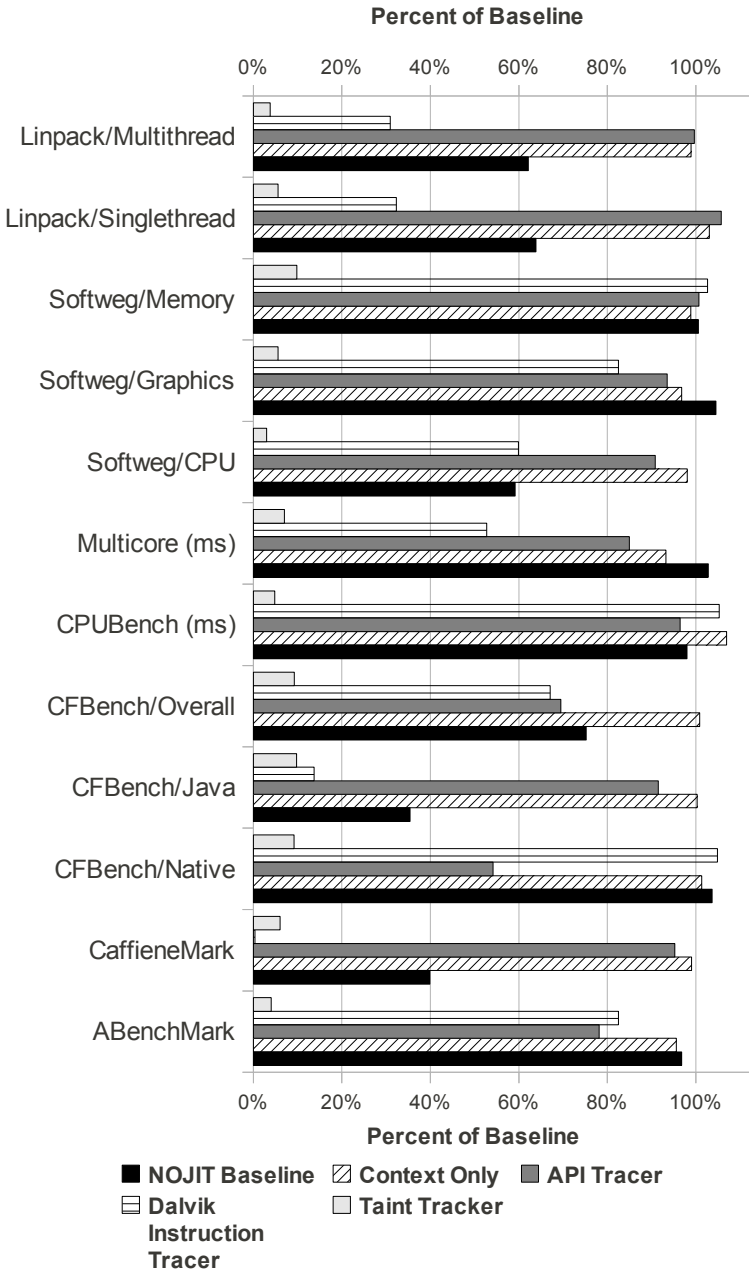

Fig. 4.7.: Benchmark Results

was configured to completely disable JIT to establish a `NOJIT` baseline. This is the only set of performance tests that required changes to the Android source. The results are summarized in the bar chart in Figure 4.7. Each tool is associated with a set of bars that shows its averaged benchmark results (y-axis) relative to the baseline as a percentage. The ARM Instruction Tracer results are excluded as they are similar to the taint tracker results.

Please note that the benchmarks are not perfect representations of performance as evidenced by the $> 100\%$ results. For example, in CPUBenchmark the standard deviation, σ, for Baseline, Dalvik tracer and Context Only is only 1%. This means that the results are consistent for each plugin, but might not be across plugins. Furthermore, the Softweg filesystem benchmarking results were removed due to high variability, $\sigma > 27\%$.

It can be seen from Figure 4.7 that the overhead (`Context Only`) of reconstructing the OS-level view is very small, up to 7% degradation. The taint tracker has the worst performance as expected, because it registers for instruction level events. The taint tracker incurs 11x to 34x slowdown, which is comparable to other taint analysis tools [56, 94] on the x86 architecture. A special case is seen in the Dalvik instruction tracer result for CaffeineMark. This result is attributed to the fact that the tracer dynamically retrieves symbol information from guest memory for logging.

The benefits of dynamically disabling JIT is evident in some Java based benchmarks such as Linpack, CFBench/Java and CaffeineMark. For those benchmarks, the API tracer's performance is greater than that of the NOJIT Baseline, despite the fact that instrumentation is taking place. This difference is due to Java libraries, such as String methods, still benefiting from JIT in the API tracer.

4.6.2 Analysis of DroidKungFu

The DroidKungFu (DKF) malware contains three components. First, the core logic is implemented in Java and is contained within the `com.google.ssearch` package. This is the main target of the investigation. Second are the exploit binaries which are encrypted in the apk, decrypted by the Java component and then subsequently executed. Third is a native library that is used as a shell. It contains JNI exported functions that can run shell commands and is the main interface for command and control. Unfortunately the command and control server was unavailable and thus this feature was not analyzed.

Discovering the Internal Logic The investigation began with the API tracer plugin. The plugin provides a high level view of how DKF interacts with the rest of the system. Behavior analysis started with a search for system calls of interest in the log. One such call is a *sys_open* for a file named "gjsvro". There was also a subsequent *sys_write* to the file from a byte array. Further analysis showed that this array is actually part of a Java ArrayObject which was populated by the *Utils.decrypt* method, which is part of DroidKungFu. Since *decrypt* takes a byte array as the parameter, a backwards search through the log revealed that this particular array was read from an asset inside the App's package file called "gjsvro". It means that during execution, DroidKungFu decrypts an asset from its package and generates the "gjsvro" file.

Subsequent entries in the log showed DroidKungFu invoking *Runtime.exec* with parameters "chmod 4755" and the name of the file, making "gjsvro" executable and setting the setuid bit. After that, *Runtime.exec* is invoked again for "su" which led to a *sys_fork*. Furthermore, the file path for "gjsvro" was then written to a `ProcessImpl OutputStream`,

```
getPermission {
  if checkPermission() then doSearchReport(); return
  if !isVersion221() then
    if getPermission1() then return
  if exists("bin/su" or "xbin/su") then
    getPermission2(); return
  if !isVersion221() then getPermission3(); return
}
```

Fig. 4.8.: getPermission Pseudocode

followed immediately by "exit". Since this stream is piped to the child's *stdin*, it is evidence that the intention of "su" was to open a shell which is then used to execute "gjsvro" followed by "exit" to close the shell. This did not work though since "su" did not execute successfully.

Given the high level view provided by the API tracer, a more detailed analysis was conducted using the Dalvik instruction tracer. The resulting trace showed that the *decrypt* and *Runtime.exec* methods were invoked from a method called *getPermission2*, which was called from *getPermission* following a comparison using the result of *isVersion221* and some file existence checks. To get a more complete picture of the *getPermission* method, dexdump was used to disassemble the class file. The overview pseudocode is shown in Figure 4.8 . The pseudocode shows that the different method invocations must be instrumented and their return values changed in order to explore the *getPermission1* and *getPermission3* methods.

With the Dalvik view support, the return values of the *isVersion221* and *exist* methods were modified and the remaining methods explored. They are essentially different ways to obtain the root privilege on different Android configurations. *getPermission1* and *getPermission2* only uses the "gjsvro" exploit. The main difference is that *getPermission1*

uses *Runtime.exec* to execute the exploit while the other uses the "su" shell. On the other hand, *getPermission3* decrypts "ratc", "killall" (a wrapper for "ratc") and "gjsvro" and executes them using its own native library. The native library's behavior can be observed from the API tracer log. The log showed the library using *sys_vfork* and *sys_execve* to execute both the "udev" and "rage against the cage" (ratc) exploits.

Analyzing Root Exploits Since Gingerbread has already been patched against these exploits, they never executed correctly. The patches were removed from the Android source and a vulnerable virtual device created to further analyze the exploits. The steps used to understanding ratc is described here, udev is analyzed in the same manner and its analysis is therefore skipped.

Once again, analysis started with the API tracer. However, no malicious behavior was evident in the log. There was some suspicious behavior in the process log provided as part of the OS-view reconstruction though. In particular, the process log showed numerous ratc processes (descendants of the original ratc process) being spawned, the *adbd* process with uid 2000 ending, followed by more ratc processes and then by an adbd process with uid 0 or root. This signifies that the attack was successful. It is worth noting that the traditional adb based dynamic analysis would fail to observe the entire exploiting process, because adbd is killed at the beginning.

Further analysis of the logs and descendent processes showed that there are in fact three types of ratc processes. The first is the original ratc process that simply iterates through the */proc* directory looking for the pid of the adbd process. Its child then forked itself until *sys_fork* returned -11 or EAGAIN. At this point it wrote some data to a pipe

and resumed forking. In the grandchild process a call to *sys_kill* is used to kill the adbd process followed by attempts to locate the adbd process after it re-spawns.

Trace-Based Exploit Diagnosis of "ratc" An example of exploit diagnosis using the ARM instruction tracer on ratc is presented in this section. These results corroborate with publicly available information on ratc and the setuid exhaustion vulnerability.

By design, *adbd* is supposed to downgrade its privileges by setting its uid to AID_SHELL (2000), and yet adbd retained its root privileges after the attack. Thus, in an effort to identify the root cause of the vulnerability, DroidScope was used to gather an ARM instruction trace that includes both user and kernel code.

A simplified and annotated log is shown in Figure 4.9. In the log, the instruction's address comes first followed by a colon, the decoded instruction and then the operands. The instructions are also indented to illustrate the relative stack depth.

The log begins when *setgid* returns from the kernel space and returns back to *adb_main* at address 0x0000c3a4. Almost immediately, the log shows *setuid* being called. After transitioning into kernel mode, *sys_setuid* is called followed by a call to *set_user*. Later an entry shows *set_user* returning an error code 0xfffffff5 which is (-11 in 2's complement or -EAGAIN).

Tracing backwards in the log revealed that this error code was the result of the *RLIMIT_NPROC* check in *set_user*. This reveals why *setuid* failed to downgrade adbd's privileges. Further analysis of the log showed that the return value from *setuid* was not used by adbd nor was a call to *getuid* seen. The same applies to *setgid*. This indicates that adbd failed to ensure that it was no longer running as root. Thus, the analysis shows that

```
  ;;;setgid returns from kernel back to adbd
  0000813c: pop {r4, r7}
  00008140: movs r0, r0
  00008144: bxpl lr : Read Oper[0]. R14, Val = 0xc3a5
  ;; Return back to 0xc3a4 (caller) in Thumb mode

;;;adbd_main sets up for setuid
0000c3a4: movs r0, #250
0000c3a6: lsls r0, r0, #3 : Write Oper[0]. R0, Val = 0x7d0
  ;; 250 * 8 = 0x7d0 = 2000 = AID_SHELL

  ...

  ;;;Start of setuid section
  ;;;  213 is syscall number for sys_setuid
  00008be0: push {r4, r7} : Write Oper[0]. M@be910bb8, Val = 0x7d0
    ;; push AID_SHELL onto the stack
  00008be4: mov r7, #213
  00008be8: svc 0x00000000
    ;; Make sys call

    ;;; === TRANSITION TO KERNEL SPACE ===

    ;;;sys_setuid then calls set_user in kernel mode

      ;;;inside sys_setuid
      ;; Has rlimit been reached?
      c0048944: cmp r2, r3  : Read Oper[0]. R3, Val = 300 Read Oper[1]. R2, Val = 300

      ;;; RLIMIT(300) is reached and !init_user so return -11
      c0048960: mvn r0, #10 : Write Oper[0]. R0, Val = 0xfffffff5
          ;; the return value is now -11 or -EAGAIN
      c0048964: ldmib sp, {r4, r5, r6, fp, sp, pc}

      ;;;Return back to sys_setuid which returns back to userspace

    ;;; === RETURN TO USERSPACE ===

  ;;;setuid continues
  00008bec: pop {r4, r7}
  00008bf0: movs r0, r0 : Read Oper[0]. R0, Val = 0xfffffff5
      ;; -11 is still here

;;;Return back to adb_main at 0xc3ac (the return address) above
;;; Immediately starts other work, does not check return code
0000c3ac: ldr r7, [pc, #356] : Read Oper[0]. M@0000c514, Val = 0x19980330
  Write Oper[0]. R7, Val = 0x19980330
    ;; 0x19980330 is _LINUX_CAPABILITY_VERSION
```

Fig. 4.9.: Annotated adbd trace

the vulnerability is due to two factors, RLIMIT_NPROC and failure to verify successful privilege downgrading.

Triggering Data leakage Reverting back to the default unchanged Gingerbread build, the Dalvik instruction tracer and taint tracker were used to understand the information leakage behavior in *doSearchReport*. As depicted in Figure 4.8, this involved instrumenting *checkPermission* during execution of *getPermission*. The Dalvik instruction trace showed *doSearchReport* invoking *updateInfo*, which obtained sensitive information about the device including the device model, build version and IMEI amongst other things. Outgoing HTTP requests, which failed because the server was down, were also observed. These HTTP requests were then redirected to a specially created HTTP server by adding a new entry into */etc/hosts*.

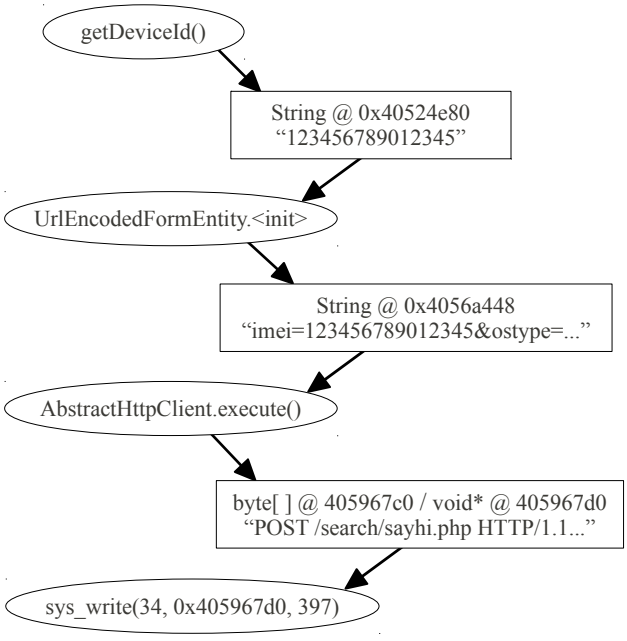

Fig. 4.10.: Taint Graph for Droid Kung Fu

The taint tracker was used to further analyze this information leakage. A simplified taint propagation graph is shown in Figure 4.10. Objects, both Java and native, are represented by rectangular nodes while methods are represented by oval nodes. The figure shows *UrlEncodedFormEntity* (the constructor) propagating the original tainted IMEI number in the String @ 0x40524e80 to a second String that looks like an HTTP request. The taint then propagated to a byte array at 0x405967c0 by *AbstractHttpClient.execute*. It finally arrived at the sink at *sys_write*. Note that *sys_write* used a void* at 0x405967d0, which is the contents array of the byte array Object (see the StringObject example in Section 4.4.2). This is expected since JNI provides direct access to arrays to save on the cost of a memory copy.

4.6.3 Analysis of DroidDream

Like analyzing DroidKungFu, the API tracer was used to get a basic understanding of DroidDream, and then instruction traces were obtained and information leakage analyzed.

The logs generated by the API tracer and the OS-view, showed two DroidDream processes. "com.droiddream.lovePositions," the main process, does not exhibit any malicious behavior except using *Runtime.exec* to execute "logcat -c" which clears Android's internal log. Again, this behavior indicates that traditional Android debugging tools fall short for malware analysis.

"com.droiddream.lovePositions:remote," the other process, is the malicious one. The logs showed DroidDream retrieving the IMSI number along with other sensitive information like IMEI, and encoded them into an XML String. Next, a failed attempt to open a

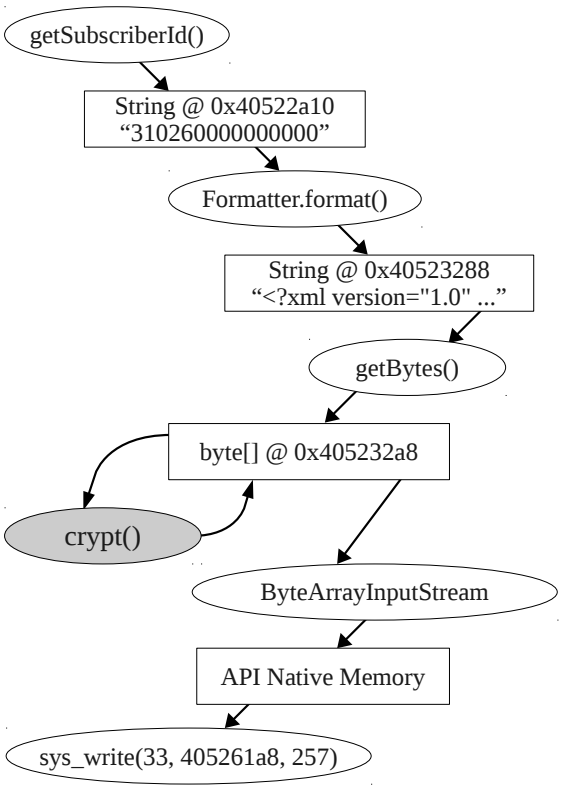

Fig. 4.11.: Taint Graph for DroidDream

network connection to `184.105.245.17:8080` was seen. This time, a different approach was used to observe the networking behavior. The return values of *sys_connect* and *sys_write* were instrumented to make DroidDream believe these network operations were successful.

Using the taint tracker, the IMSI was marked tainted and a taint propagation graphs was obtained, which confirm that DroidDream did leak sensitive information from these sources to a remote HTTP server. The simplified graph for leaking IMSI information is illustrated in Figure 4.11. The graph was also annotated to include *crypt* which is the DroidDream method used to xor-encrypt the byte array. The graph shows that *getSubscriberId* was used to obtain the IMSI from the system as a String @ 0x40522a10. The IMSI String, along with other information, were then encoded into an XML format

```
[0x43328f40] aget-byte v2(0x01), v4(0x405232a8), v0(186)
  Getting Tainted Memory: 0x40523372(0x2401372)
  Adding M@0x410accec(0x42c5cec) len = 4
[0x43328f44] sget-object v3(0x0000005e), KEYVALUE// field@0003
[0x43328f48] aget-byte v3(88), v3(0x4051e288), v1(58)
[0x43328f4c] xor-int/2addr v2(62), v3(41)
  Getting Tainted Memory: 0x410accec(0x42c5cec)
  Adding M@0x410accec(0x42c5cec) len = 4
[0x43328f4e] int-to-byte v2(0x17), v2(23)
  Getting Tainted Memory: 0x410accec(0x42c5cec)
  Adding M@0x410accec(0x42c5cec) len = 4
[0x43328f50] aput-byte v2(17), v4(0x405232a8), v0(186)
  Getting Tainted Memory: 0x410accec(0x42c5cec)
  Adding M@0x40523372(0x2401372) len = 1
```

Fig. 4.12.: **Excerpt of Dalvik Instruction Trace for DroidDream.** A Dalvik instruction entry shows the location of the current instruction in square brackets, the decoded instruction plus the values of the virtual registers in parenthesis. A taint log entry is indented and shows tainted memory being read or written to. The memory's physical address is shown in parenthesis and the total bytes tainted is represented by "len."

using *format*. The resulting String was then converted into a byte[] @ 0x405232a8 for encryption by *crypt*. The encrypted version was used to create a *ByteArrayInputStream*. For brevity, a generic "API Native Memory" node is used to illustrate the taint further propagating through memory until the eventual sink at *sys_write*.

The *crypt* method was further investigated by augmenting the Dalvik instruction tracer to track taint propagation and generate a taint-annotated Dalvik instruction trace. Not only did the log entries show the byte array being xor-ed with a static field named "KEYVALUE," they also showed that the encryption was conducted on the byte[] in-place. A snippet of the trace log is depicted in Figure 4.12.

DroidDream also includes the udev and ratc exploits (unencrypted), plus the native library terminal like DroidKungFu. They were not further analyzed.

4.7 Discussion

Limited Code Coverage Dynamic analysis is known to have limited code coverage, as it only explores a single execution path at a time. To increase code coverage, one may explore multiple execution paths as demonstrated in previous work [39–41]. Alternatively, the experiments demonstrated that different paths can be explored by manipulating the return values of system calls, native APIs and even internal Dalvik methods of the App. This simple approach worked well, although a more systematic approach is desirable. One method is to perform symbolic execution to compute path constraints and then automatically explore other feasible paths. DroidScope does not yet support symbolic execution and it is left as future work.

Detecting and Evading DroidScope As with other emulation based malware analysis platforms, transparency is an issue with DroidScope. More troubling are the intrinsic differences between the emulated environment and mobile systems. Mobile devices contain numerous sensors (e.g., GPS, motion and audio), with performance profiles which might be difficult to emulate. While exploring multiple execution paths may be used to bypass these types of tests, they might still not be sufficient. For example it was observed that Android, as an interactive system, can be sensitive to the performance overhead due to analysis. If the analysis takes too long, certain timeout events are triggered leading to different execution paths or even the killing of the process being analyzed. The analyst must be aware of these new challenges. Heterogeneous record and replay might be a potential solution to this new class of transparency problems, but further investigation in this area is needed.

4.8 Conclusion

This chapter showed that the two levels of semantic information in Android platforms can be effectively reconstructed from outside the virtual machine and seamlessly bound to the execution so that cooperating Java and native components can be analyzed using one single platform. Not only that, but this chapter also discussed the techniques used to build a new, highly dynamic, emulation based malware analysis platform for analyzing Android malware so that the advantages of flexibility and efficiency are preserved. The techniques were implemented in DroidScope and verified by analyzing real-world malware. In short, emulation based Android malware analysis is feasible.

With the results discussed in V2E and new ARM architectures supporting hardware virtualization, it is foreseeable that DroidScope can achieve the same transparency properties afforded by V2E. What remains of this dissertation is a fundamental understanding of dynamic taint analysis so that its precision can be improved. Additionally, the new found understanding will be used to determine whether DroidScope's assumption of tracking taint at the native instruction level, (e.g., ARM), is sufficient for tracking information flows in both native and Java code is true.

5. UNDERSTANDING DYNAMIC TAINT ANALYSIS

5.1 Introduction

Both V2E and DroidScope support dynamic taint analysis, albeit with slightly different designs. What remains unknown is whether the propagation policies of these and other taint analysis implementations are accurate and, if so, how precise. This chapter presents results towards a better understanding of how the different design parameters of taint-granularity, analysis-granularity and special case support affect false-positives and false-negatives. Furthermore, methods for verifying the accuracy and precision of implementations are also presented.

In an effort to focus the discussion, the models, definitions and examples will be framed around taint analysis for bitvector machines. That is, the data items that are labeled tainted or untainted can be represented as bitvectors, and the operations that propagate the taints have bitvector operands. Both x86 and ARM are bitvector machines. While the discussions are limited to bitvectors, the general concepts and observations are not. Applications of the results to other machines, such as Java, are discussed briefly.

This chapter starts with the derivation of a model for analyzing the accuracy and precision of taint propagation policies from the formal model of noninterference. The commonly used terms of *over-taint*, *under-taint*, *false-positive*, *false-negative*, *accurate* and *precise* are also defined against this model. These details are presented in Section 5.2. The

relationship between the design parameters of taint-granularity, analysis-granularity and special cases support and information flow is also discussed in the same section.

The sources of false-positives and false-negatives in practice, including the fundamental relationships between the design parameters and false-positives, are discussed in Section 5.2.2. In brief, a taint analysis implementation with a coarser taint-granularity (e.g., 32 bit word-level) can have more false-positives than an implementation with a finer taint-granularity (e.g., byte-level). Similarly, propagating taint using an IR might also have more false-positives than propagating taint through the native instructions. The methods presented in this chapter can be used to verify that there are no false-positives.

To illustrate the benefits of a formal based approach, the information flow and verification problems are formulated as satisfiability problems that can be solved using Satisfiability Modulo Theory (SMT) solvers. The Z3 SMT solver [129] is then used to automatically generate a taint propagation policy that is guaranteed not to have any false-negatives. This is done by first defining the behavioral semantics of 23 x86 instructions, and then using Z3 to calculate the information flows from input operands to output operands at the bit level. The details are presented in Section 5.4.

In order the assess the quality of the automatically generated policy, it is scrutinized by itself and then compared to the policies used in previously published taint analysis platforms in terms of false-positives and false-negatives. Additionally, since some implementations have special rules for reducing false-positives, they are verified for correctness using the formal model. Comparison and verification are discussed in Section 5.5. Also discussed in the same section is DroidScope's assumption that Dalvik-level

tainting is unnecessary(Section 5.5.5). Finally, limitations are presented in Section 5.6 and conclusions drawn in Section 5.7.

5.2 Formal Foundation

This section begins with the derivation of the formal model used to analyze taint propagation policies. This model is used to analyze the information flows between data items (e.g., input and output operands), and is based on noninterference. Once the model is established, observations on how the design parameters of taint-granularity, analysis-granularity and special cases support can be mapped to model are made. Additionally, the taint analysis challenges outlined in the Background Chapter - false-positives, false-negatives, sanitization, and implicit flows - are also mapped to the formal model. In short, these challenges arise from the definition of noninterference; there are fundamental limits to how precise a taint propagation policy can be.

This section concludes with a description of what a taint policy is and what it should be. In particular, a "golden policy" is defined to enforce the noninterference property exactly. This golden policy is then used to define the common terms of over-tainting, false-positives, etc.

5.2.1 Noninterference

Noninterference was first described by Goguen and Meseguer [20] to analyze the information flows between users in a multi-user system. They defined a machine M that consists of:

$$S: \text{ the set of machine states}$$

$$s_0: \text{ the initial state}$$

$$U: \text{ the set of users}$$

$$C: \text{ the set of commands}$$

$$(U \times C): \text{ the set of inputs}$$

$$(U \times C)^*: \text{ the set of input sequences}$$

For simplicity, $(U \times C)^*$ is called a *program* and a sequence $w \in (U \times C)^*$ an *execution path* through the program. A sequence w is executed through the repeated application of the state transition function *do:* $S \times U \times C \rightarrow S$ until all commands are processed. Intuitively, there is no information flow from a sending user u_s to a receiving user u_r (i.e., the sending user is noninterfering with the receiving user), if and only if the outputs observed by u_r are not affected by the actions of user u_s after executing the program starting from an initial state s_0.

Goguen and Meseguer expressed this formally by defining $[[w]]_u$ as the output seen by the user $u \in U$ after w has been processed using the *do* function and $P_X(w)$ as the subsequence of w where all commands by users $X \subseteq U$ have been purged (removed).

Hence, for an initial state s_0, a set of users G is noninterfering with a second set of users G' if and only if for all w in $(U \times C)^*$ and all u in G', $[[w]]_u = [[P_G(w)]]_u$.

Since taint tracking is designed to analyze the information flow between two data items and not users in the system, the noninterfering data problem is mapped into a noninterfering users problem by coupling data items to users. M' is used to represent the noninterfering data machine. Also, the 32-bit x86 instruction `add dst, src`, is used as an example to illustrate the core concepts. That machine will be signified by the + superscript and framed for emphasis.

First, S' is defined as the Cartesian product of the set of states for each individual data item S_i. Thus, S' is a bitvector that is composed of other bitvectors. "Data item" is used as a generic term for the machine's memory and registers. Hence, given n data items:

$$S' = S = S_1 \times S_2 \times S_3 \times ... \times S_n$$

For brevity, a set of aliases $\hat{S} = \{1..n\}$ such that $i \in \hat{S}$ will be used to refer to the i^{th} data item is used. Alias sets $\hat{S_1}..\hat{S_n}$ are used to refer to the individual bits of the data items. That is, $j \in \hat{S_i}$ is used to refer to the j^{th} bit of the i^{th} data item.

The `add` instruction only has two operands, `dst` and `src`, with `dst` being both an input and an output. The rest of the state is ignored since those data items are independent of the `add` instruction (i.e., they do not affect the operation of the instruction). Thus, for the example, S^+ can be simplified as:

$$S^+ = S_{dst} \times S_{src}$$

This item based definition can be used to determine if there is information flow between data items $s, r \in \hat{S}$. This is done by coupling s with a sending user u_s, r with a receiving user u_r and the processing of the program with a processing user u_p. In essence, u_s communicates with u_r, who can only observe the output value of r, by manipulating the initial value of s. Thus, there is information flow from s to r in M if and only if there is information flow from u_s to u_r in M'. Consequently, there are three users with all commands assigned to u_p:

$$U' = \{u_s, u_r, u_p\}$$

$$(U \times C) = (u_p \times C)$$

$$(U \times C)^* = (u_p \times C)^*$$

The example consists of a single command, `add`, and in instruction-level tainting (the current focus) `add` is a simple operation. If an IR was used to emulate the instruction, then the commands will contain the full IR instruction set and there will be longer and more input sequences as well. Informally, $\leftarrow$ is used as an operator that updates the state of the left operand with the value of the right, and $add(dst, src)$ is used to represent the function that sums dst and src. Subsequently, the commands and sequences can be defined as:

$$C^+ = \{dst \leftarrow add(dst, src)\}$$

$$(U \times C^+) = \{(u_p, dst \leftarrow add(dst, src))\}$$

$$(U \times C^+)^* = \{(u_p, dst \leftarrow add(dst, src))\}$$

To allow the sending user to manipulate the initial state or value of the sending data item s, a value setting command, $v \in V$, is prepended to $(U \times C)^*$ as represented by the $\cdot$ operator. Purging the sending user's commands results in the original program.

$$V : \text{the set of commands to assign values to } S_s$$

$$C' : V \cup C$$

$$(U \times C') = (u_s \times V) \cup (u_p \times C)$$

$$(U \times C')^* = \{(u_s, v) \cdot w | v \in V, w \in (U \times C)^*\}$$

In the example, assume that the information flows from `src` to `dst` are to be determined. In this case, the receiving user will monitor the output of the operation (i.e., `dst`), and the sending user will manipulate the `src` operand.

$$[[w]]^+_{u_r} = [[w]]_{dst,s_0}$$

$$V^+ = \{src \leftarrow i \mid i \in \{0..2^{32} - 1\}\}$$

$$(U \times C^+) = \{(u_s, src \leftarrow 0), (u_s, src \leftarrow 1), ...$$

$$, (u_p, dst \leftarrow add(dst, src))\}$$

$$(U \times C^+)^* = \{(u_s, src \leftarrow 0) \cdot (u_p, dst \leftarrow add(dst, src))$$

$$, (u_s, src \leftarrow 1) \cdot (u_p, dst \leftarrow add(dst, src))$$

$$, ...\}$$

Consequently, by prepending inputs (u_s, v) to $(U \times C)^*$, $(U \times C')^*$ is the set of new sequences of inputs where the initial value of s is different in each case. Then, by defining $G = u_s$ and $G' = u_r$, the purge $P_G(w \in (U \times C')^*)$ results in the original sequences of

commands $(U \times C)^*$ in M. Given this setup and by the definition of noninterference, u_s is noninterfering with u_r if and only if

$$\forall w \in (U \times C')^*, \qquad [[w]]_{u_r} = [[P_G(w)]]_{u_r}$$

Since there is a $v \in V$ that assigns the initial value of s back into s (i.e., the value of s was not changed by the input (u_s, v)), purging this value setting input has no effect on the output value of r. Thus, the condition above simplifies into:

$$\forall w, w' \in (U \times C')^*, \qquad [[w]]_{u_r} = [[w']]_{u_r}$$

If the condition is satisfied, then the final value of r is independent of the initial value of s. This is the definition for noninterfering data items. Consequently, the inverse relationship is used to determine if there is information flow between two data items. There is information flow from $s \in \hat{S}$ to $r \in \hat{S}$ by the sequence of inputs $(U \times C)^*$ with initial state s_0 if and only if $\exists w, w' \in (U \times C')^*$ such that $[[w]]_{u_r} \neq [[w']]_{u_r}$.

For the bitvector machine, information flow can be further refined as a function of the initial and end values of the sending and receiving data items and thus arriving at Definition 5.2.1. The definition uses some additional notation which are introduce first.

The following description uses C-style bitvector operators. First $Val(x, y)$ is a function that maps the data item $x \in \hat{S}$ and value $y \in S_x$ to a bitvector of the same length as s_0 with bits corresponding to item x set to y. All other bits are 0. For the example, $Val(src, 0x00001234) \rightarrow 0x0000000000001234$. Then, $Mask(x) = Val(x, Ones(x))$ where

$Ones(x)$ is a bitvector of the same length as x with all bits assigned to 1. For the example, $Mask(src) \rightarrow$ 0x00000000ffffffff. Finally $s_0(x, y)$ denotes $(s_0 \& (Mask(x)))|Val(x, y)$, where the value of x has been changed to y in s_0.

The notation $[[w]]_{r,s_0(s,x)}$, with $s, r \in \hat{S}$ and $x \in S_s$, is used as the output value of data item r after processing a sequence w with initial state s_0 where the value of data item s has been changed to x. For example if $S = S_1 \times S_2$, with $S_1 = S_2 = \{0, 1\}$ and $s_0 = (0, 0)$, then $s_0(1, 1) = (1, 0)$ and $s_0(2, 1) = (0, 1)$.

Definition 5.2.1. *For $s, r \in \hat{S}$, there is information flow from s to r, represented by $s \rhd_0 r$, if and only if $\exists w \in C^*$, $[\exists i, j \in S_s$ s.t. $[[w]]_{r,s_0(s,i)} \neq [[w]]_{r,s_0(s,j)}]$*

Definition 5.2.1 can be read as: for an initial state s_0 there is information flow from s to r if and only if there exists two values of s such that the values of r differ after processing any path of the program. The users u_s and u_r have been conveniently removed and as a result the machine model for taint analysis only consists of S, s_0, C and C^* from the original information flow machine model M. This shorter notation will be used in this rest of this chapter.

In the example, given $s_0 = (dst_0, src_0)$, $[[w]]_{r,s_0(s,i)} = add(dst_0, i)$. Then by Definition 5.2.1, it is obvious that there is information flow from *src* to *dst* since one can choose $i = 0x00000000$ and $j = 0x00000001$ so that $add(dst_0, i) \neq add(dst_0, j)$. This relationship is not always so simple in practice though. Dynamic information trackers often have to make certain design trade-offs to achieve different goals such as performance, storage overhead and general applicability. Four major design parameters are highlighted in the observations below.

Observation 5.2.1. *The data items were defined as the operands* `dst` *and* `src`*; however, they can be defined as small as a single bit. This design parameter of choosing the data item size is known as the* taint-granularity *and has implications on storage overhead and performance.*

For example, if byte-level tainting was used, then the set of states for *dst* will be a product of four smaller sets, one for each byte: $S_{dst} = S_{dst_3} \times S_{dst_2} \times S_{dst_1} \times S_{dst_0}$. The same applies to S_{src}. In this case, there is no information flow from src_0 to dst_1 if the initial state was $s_0 = (s_{dst_3}, s_{dst_2}, s_{dst_1}$,0x0000$, s_{src_3}, s_{src_2}, s_{src_1}, s_{src_0})$.

Observation 5.2.2. *In the example,* `add` *is a basic or atomic instruction. What if the instruction was more complex such as Bit-Scan-Forward (*`bsf`*)? Since* `bsf` *iterates through the bit positions in search of the first 1-bit, C^* is expected to contain multiple sequences of basic instructions. In this case, how is taint to be analyzed? Should* `bsf` *be analyzed as an atomic instruction or should its taint propagation behavior be composed using those of the basic instructions? This design parameter of choosing the smallest executable unit to analyze is the* analysis-granularity*.*

By definition, information flow is analyzed for the program as a whole; however, most taint analysis implementations analyze information flow for each command in C and then sequentially apply them to the individual inputs of the sequences in C^*. IR based tainting is an application of this concept. For example TEMU [16] propagates taint through QEMU's internal IR instead of the native x86 instructions. Thus, one or more IR instructions is used to emulate an x86 instruction. The taint propagation policy for x86 instructions are effectively simple compositions of the taint propagation policy for IR

instructions. Tiwari et al. pointed out that false-positives can result from simple composition in gate-level tainting [108].

Observation 5.2.3. *Definition 5.2.1 is* initial state aware*, that is, information flow must be analyzed for each and every initial state separately, an unlikely possibility in dynamic taint analysis due to performance constraints. In practice, taint propagation rules are* initial state agnostic *or state agnostic for short.*

Due to the desire to minimize runtime overhead, taint propagation policies are commonly defined off-line with one rule summarizing the flows for all initial states. This is one fundamental source of the need for sanitization. For the $b = a \oplus a$ example, initial state aware analysis will reveal that there is no information flow from a to b for this particular set of initial states. Reflecting the state of practice, $S_{\triangleright}$ is used to denote the set of initial state agnostic flows for C^* and is defined below. State awareness is a part of the special cases design parameter described previously.

Definition 5.2.2. *Using $S_{\triangleright_i}$ as the set of all information flows for a program with an initial state s_{0_i} - i.e., for a program C^*, $S_{\triangleright_i} = \{(s,r) | s,r \in \hat{S}$ and $s \triangleright_0 r$ with initial state $s_0 = i \in S\}$ - $S_{\triangleright} = \bigcup S_{\triangleright_i}$. That is, $S_{\triangleright}$ is the set of all information flows.*

Since $S_{\triangleright}$ is initial state agnostic, it specifies that there is information flow from s to r as long as there is an initial state such that there is information flow from s to r. It is clear that the set union can be overly conservative and introduce false-positives.

Observation 5.2.4. *The conservative nature of noninterference states that there is information flow through a program as long as there is flow through one of its paths.*

As dynamic taint analysis can only "see" the path that is executed, this is a fundamental problem that can't be addressed through pure dynamic taint analysis.

5.2.2 Taint Propagation Policies

A taint propagation policy is a set of rules that determine when and how a data item should be labeled or tainted. Since taint analysis is only concerned with information flows from tainted data items, the rules are predicated on the fact that the flow source is tainted. This section begins with the definition for a "golden" policy and then delves deeper into the sources of over- and under-tainting.

Definition 5.2.3. *For a particular choice of taint- and analysis-granularity, a "golden" taint propagation policy T_G is one that propagates taint exactly when there is information flow from a tainted source and clears the taint otherwise. For $s, r \in \hat{S}$, $(s, r) \in T_G$ if and only if s is tainted and $s \rhd_0 r$*

Ideally, taint analysis platforms will use the golden policy that is defined at the most descriptive taint- and analysis-granularities. Jif [130], based on the JFlow language [131], and FlowCaml [132] are examples of tools that analyzes information flow through whole Java and ML programs respectively and applies the most appropriate golden policies. They are both static analysis tools though. Using the perfect golden policy is impractical for dynamic analysis. First, dynamic analysis is limited to analyzing the single path that is actually executed. This leads to the loss of implicit control flow information since they are due to paths that are not executed (a problem related to Observation 5.2.4). Second, the taint-granularity is set to balance precision and performance in practice.

Third, and perhaps more importantly, information flow is often tracked based on a common set of low level commands (e.g., x86 instructions), and not the operations in which the program's logic was expressed. This decision is likely dictated by the need to analyze programs whose components are written in different languages. For example, none of the implementations surveyed propagates taint at the C++ level for a C++ program, at the C level for standard libraries and at the x86 level for assembly code at the same time. As will be discussed next, these practical choices in taint- and analysis-granularity can lead to false-positives. Furthermore, the problem can surface in both dynamic and static analysis tools if information flow is analyzed using an IR.

5.2.3 Over- and Under-tainting

In practice, taint propagation policies are only approximations of the corresponding golden policy (i.e., the if and only if relationship is relaxed). Thus, there is a need to objectively compare different policies. To this extent, the terms *false-positive*, *false-negative*, *accuracy*, *precision* and *over-taint* are defined against a common standard. For the purposes described in this chapter, the golden policy T_G is used as the standard. This implies that two policies compared using the definitions have the same taint- and analysis-granularities or are at least comparable. Given the spirit of definitions, the golden policy can also be defined based on an enumeration of all the acceptable false-positive and false-negative cases. Effectively, this is the approach taken by most implementations. However, this type of standard is fragile since what is acceptable can change.

Definition 5.2.4. *Given a policy T, (s, r) is a* false-positive *if $((s, r) \in T \wedge (s, r) \notin T_G)$ and (s, r) is a* false-negative *if $((s, r) \notin T \wedge (s, r) \in T_G)$.*

Definition 5.2.5. *A policy T is* accurate *if it does not contain any false-negatives and it is* precise *if it is accurate and does not contain any false-negatives.*

Definition 5.2.6. *Given two policies, T_A and T_B, T_A* over-taints *T_B if*

$\exists\ (s, r),\ (s, r) \in T_A \wedge (s, r) \notin T_B$.

Definition 5.2.7. *Given two policies, T_A and T_B, T_A* strictly over-taints *T_B if $T_A \supseteq T_B$.*

Under-tainting is defined similarly to over-tainting and strictly over-taints is a new term introduced to emphasize a special relationship. If neither T_A nor T_B contain any false-negatives (i.e., are accurate), then T_A over-taints T_B means T_A has false-positives that T_B does not. Furthermore, if T_A strictly over-taints T_B, then T_B is more precise than T_A unless they are equal. In this case, the terms over-taints, more precise, and more false-positives than can be used interchangeably.

Additionally, the pessimistic nature of noninterference ensures that if the policies are based on or are verified against Definition 5.2.1 (i.e., $(s, r) \in S_{\triangleright_0} \implies (s, r) \in T$), then there are no false-negatives. This concept is used as the basis for analyzing the precision of previously published taint analysis platforms. The golden policy is also used as the basis for discussing sources of over- and under-tainting in the next section.

5.3 Sources of Over- and Under-tainting

The observations made in the previous section suggested some fundamental sources of false-positives and negatives in dynamic taint analysis platforms. This section focuses on

showing how the design choices of taint-granularity, analysis-granularity and special cases can impact precision.

5.3.1 Over-tainting Due to Taint-granularity: Observation 5.2.1

The minimum data item that can be labeled as tainted is known as the *taint-granularity*. In many applications, the taint granularity is set at the byte-level to match byte-level memory addressing. This provides good coverage for memory data and has been effective as evidenced by the numerous published applications of byte-level tainting. On the other hand, bit-level tainting has also been used. In Memcheck [84] for example, bit-level tainting is used because the additional precision reduces the false positive rate for detecting memory errors.

This additional precision can be illustrated using the shift left instruction `shl dst, imm8` that shifts the `dst` register `imm8` positions to the left. In byte-wise tainting, if the least significant byte of `dst` is tainted, and `imm8` is 1 then the second least significant byte of `dst` is subsequently tainted. Alternatively, if only bit 0 of the least significant byte is tainted, then only bit 1 is tainted under the bit-wise policy. In the byte-wise policy, the second byte must also be tainted to ensure accuracy.

While there are many more examples and the relationship seems intuitive, a proof for the general case is provided below. Theorem 5.3.1 states that given two machines M^{byte} and M^{bit} with the only difference being the composition of the state S, the byte-level golden policy will strictly over-taint the bit-level golden policy.

Theorem 5.3.1. *$T_{bit} \subseteq T_{byte}$, i.e., the golden policy for byte-wise tainting strictly over-taints the golden policy for bit-wise tainting.*

Proof The details of the two machines are skipped over for brevity. It is simply assumed that the commands, inputs and initial states of the two machines are equivalent. Also, subscripts are used to distinguish between the two machines only when necessary. For example, they are not used for C^* since the set is common to both machines.

To prove that T_{byte} strictly over-taints T_{bit}, two functions are defined first: a function $bits(x)$ that maps a byte $x \in \hat{S}_{byte}$ to the corresponding set of bits $X' \subseteq \hat{S}_{bit}$ and the reverse function $byte(x')$ that maps a bit $x' \in \hat{S}_{bit}$ to its corresponding byte $x \in \hat{S}^{byte}$ such that $x' \in bits(x)$. It then follows that given $s = byte(s')$ and $r = byte(r')$, to show $T_{bit} \subseteq T_{byte}$, it must be shown that

$$\forall s', r' \in \hat{S}_{bit}, (s', r') \in T_{bit} \implies (s, r) \in T_{byte} \tag{5.1}$$

By Definitions 5.2.1 and 5.2.3, $(s, r) \in T_{byte}$ if and only if $\exists w \in C^*$ s.t. $\exists i, j \in S_{s_{byte}}$ where $[[w]]_{r,s_0(i)} \neq [[w]]_{r,s_0(j)}$. Then, since bytes x and y are different if and only if at least one of their bit values differ, the inequality can be rewritten and Equation 5.1 becomes:

$$\begin{aligned}
&\forall w' \in C^*, \exists w \in C^* \text{ s.t.} \\
&\quad [\exists i', j' \in S_{s'} \ , \ [[w']]_{r',s_0(s',i')} \neq [[w']]_{r',s_0(s',j')}] \implies \\
&\quad [\exists i, j \in S_s \ , \ \exists b' \in bits(byte(r)) \text{ s.t.} \\
&\qquad [[w]]_{b',s_0(s,i)} \neq [[w]]_{b',s_0(s,j)}]
\end{aligned} \tag{5.2}$$

Equation 5.2 is true since one can always choose $w = w'$, $b' = r'$ and i and j where the s' bit of the byte s is assigned values of i' and j'.

The same logic can be used to prove that, in general, coarse-grained taint policies will strictly over-taint finer-grained policies. The underlying intuition is similar to Observation 5.2.3. In order to maintain accuracy, the coarse-grained policy must propagate taint if any of its finer-grained counterparts propagate taint. Once again, the union and existential quantifiers are over-cautious and lead to false-positives.

5.3.2 Analysis-granularity and Over-tainting: Observation 5.2.2

Similar to taint-granularity above, there is also evidence that suggests *analysis-granularity* affects false-positives. Take the following C statement for example: *y = x &˜x;*. It is evident that there is no information flow from x to y since y is always equal to 0. This result is reflected if the statement is analyzed as a whole. On the other hand, if the ˜ and & operations were analyzed sequentially, then y becomes tainted if x is tainted, a false-positive.

Understanding the fundamental over-tainting relationship between instruction-level and basic-block-level information flow tracking is the focus of the following discussion. By definition, a basic block is a block of instructions with a single entry and a single exit. It is a single linear sequence of inputs $c_1, c_2, c_3, ..., c_n$.

In the per-instruction case, s_0 is defined as the initial state before the sequence of inputs is processed, s_i as the intermediate state of the machine after inputs c_1 through c_i are processed, s_n as the final state after it is processed and T_{c_i} as the taint propagation

policy for instruction c_i. To analyze the information flow between s and r using intermediary data items r_i assume that there is information flow between s and r if and only if there is a sequence of intermediary flows through r_i.

$$\begin{aligned}(s,r) \in T_c \iff & \exists (r_1, r_2, r_3, ..., r) \text{ s.t.} \\ & (s, r_1) \in T_{c_1} \wedge (r_1, r_2) \in T_{c_2} \\ & \wedge (r_2, r_3) \in T_{c_3} \wedge ... \wedge (r_{n-1}, r) \in T_{c_n}\end{aligned} \tag{5.3}$$

Intuitively, this is true since the taint must propagate from data item s to data item r through intermediary data items r_i. Furthermore, since there are n^i different paths through different intermediary items, with n being the number of data items in S, there is information flow as long as a single path exists. Once again, information flow can be over-cautious. The following theorem states that the golden policy based on per-instruction tainting strictly over-taints the one based on basic block tainting.

Theorem 5.3.2. *$T_{bb} \subseteq T_c$, i.e., per-instruction tainting strictly over-taints basic block tainting.*

Proof Induction is used to prove this theorem.

Basis: The base case with only one single instruction is clear since the two policies are equal.

Induction: For the induction case, assume that the relationship holds for i instructions - i.e., $(s, r_i) \in T_{bb}$ and Equation 5.3 is true. What is left is to show that it also holds for $i+1$ instructions by contradiction.

The $i+1$ case being false means that $(s, r_{i+1}) \in T_{bb}$ but there is no sequence such that Equation 5.3 is satisfied. Given that the i case is true, this means that for all possible sequences, only the final case of $(r_i, r_{i+1}) \in T_{c_{i+1}}$ is false. It being false means that r_{i+1} is a constant, and thus $(s, r_{i+1}) \notin T_{bb}$ since Definition 5.2.1 is no longer satisfied. This is a contradiction and therefore the original implication must hold. In general, analyzing information flow at the instruction-level will strictly over-taint as compared to analyzing at the basic block level.

Since in the best case scenario, a single basic block of IR is used to emulate a single instruction, this theorem also implies that propagating taint through the IR strictly over-taints propagating taint through the basic block and equivalently the instructions being emulated. Thus, if the IR level policy is accurate, there will only be false-positives. In the case where the emulation code includes control flows (e.g., the `bsf` instruction), Theorem 5.3.2 together with Volpano's result [112] indicate that IR level tainting not only contains false-positives, but it can also contain false-negatives as summarized in Lemma 5.3.1.

Lemma 5.3.1. *IR level tainting both over-taints and under-taints instruction level tainting.*

As a result, if precision is desired, care must be taken to verify that the information flow patterns of the two different designs are the same (at least for the operations of interest). In other words, T_A and T_B are *tainting equivalent* if $T_A = T_B$. Verification is discussed next.

5.3.3 Other Sources of Over- and Under-tainting

The previous section presented fundamental results on how the design decisions of taint- and analysis-granularity affect precision. Unfortunately over-tainting can still be a problem even between policies using the same taint- and analysis-granularities due to the initial state agnostic nature of dynamic taint analysis. These are commonly identified as special cases and handled by defining special taint propagation or sanitization rules.

There are two types of special cases: *concrete values* and *identity relationships*. Concrete value special cases were outlined in Observation 5.2.3 and result from the practice of pre-calculating the initial state agnostic taint propagation policy $S_{\triangleright}$ (Definition 5.2.2). Whereas identify relationships are special cases due to specific relationships between operands for a particular operation irrespective of the value assignments themselves. Identity relationships are sets of concrete values with special properties.

The `mul rm32` instruction, where `edx:eax = eax * rm32`, can be used to illustrate over-tainting due to a concrete value. In general, both `eax` and `edx` will be tainted after execution if `eax` was tainted before. Thus, this is likely to be the generalized taint propagation rule. An exception to this generalization is if `rm32 = 0` though. That is, if the initial state s_0 has `rm32 = 0` then the condition in Definition 5.2.1 will be unsatisfiable for $s \in \hat{S}_{eax}$ and $r \in \hat{S}_{eax}$. `eax` always equals 0. This special case is only valid for this particular value assignment of `rm32`. On the other hand, `xor eax, eax` will always result in `eax` being 0 irregardless of what the initial value of `eax` is. It is an example of an identity relationship.

Verifying Special Cases Special cases are often identified manually and then incorporated into taint propagation policies as special rules (e.g., sanitization rules) to reduce over-tainting as the need arises. These rules are then verified manually or simply left unverified due to the difficulties of manual verification. For example, Memcheck has the most special case rules, but according to its project suggestions webpage, formal verification of the rules is still needed [133]. The concepts for formal verification are introduced in this section.

The special rules in Memcheck, like many others, are defined as functions on shadow taint variables and thus verification is discussed in this specific context. These shadow taint variables are encapsulated in a shadow state S_t (with a corresponding alias set $\hat{S\_t} = \hat{S}$) that is used to label whether the data items in S are tainted. In this way, S_t is a bitvector with length equal to the number of data items in S. The taint propagation policies can then be thought of as rules or functions that transform the shadow state. A *rule* is therefore a relation $rule : S \times S\_t \times C^* \rightarrow S\_t$. Effectively, these rules summarize the set $S_{\triangleright}$.

For $r \in \hat{S}$, the transformation can be defined as:

$$S\_t_r = \begin{cases} 1 & \exists s \in \hat{S} \text{ s.t. } S\_t_s = 1 \wedge (s, r) \in T \\ 0 & \text{otherwise} \end{cases}$$

The above equation states that data item S_r tainted if and only if information flowed from a tainted data item S_s to it. It then follows that special case analysis based on the transformations of the shadow taint variables can be formally verified by ensuring that for every bit r of S_t that is zero after the transformation, the value of S_r must be a constant

for all assignments of tainted data items S_s without changing the non-tainted items. In other words, if a data item r is not tainted after processing a sequence, then all of the tainted items before processing the sequence are noninterfering with S_r.

Based on this logic, the condition for checking for false-negatives is presented in Equation 5.4 and the definition in Definition 5.3.1. Similarly Definition 5.3.2 and Equation 5.5 can be used to check for false-positives. The notation in these definitions are changed so they are closer to instruction level taint analysis, which is discussed in the next section. $x \in S$ and $y \in S$ are used as the machine states before and after processing an instruction or operation, op, where $y = op(x)$. x_t and y_t represent the corresponding shadow taint states and $y\_t = rule(x, x\_t, op)$ is the transformation function.

Definition 5.3.1. *A rule $y\_t = rule(x, x\_t, op)$ is accurate (i.e., does not contain false-negatives), if for all assignments of x and x_t the following condition is true.*

$$\begin{aligned} &\forall\ r \in \hat{S\_t}, [(y\_t_r = 0)\ \Longrightarrow \\ &\quad \forall\ j, k \in S, [\ op((x\ \&\ (\sim x\_t))|(j\ \&\ x\_t))_r = \\ &\quad op((x\ \&\ (\sim x\_t))|(k\ \&\ x\_t))_r]\ ] \end{aligned} \tag{5.4}$$

Definition 5.3.2. *A rule* $y\_t = rule(x, x\_t, op)$ *does not contain false-positives if for all assignments of* x *and* x_t *the following condition is false.*

$$\begin{aligned} &\exists\ s \in \hat{S\_t}, [\ (y\_t_r = 1)\ \wedge \\ &\quad \forall\ j, k \in S[\ op((x\ \&\ (\sim x\_t))|(j\ \&\ x\_t))_r = \\ &\quad op((x\ \&\ (\sim x\_t))|(k\ \&\ x\_t))_r]\ ] \end{aligned} \tag{5.5}$$

Definition 5.3.3. *The transformation function* $y\_t = rule(x, x\_t, op)$ *is precise if for all assignments of* x *and* x_t *Equation 5.4 is true and Equation 5.5 is false (i.e., Definitions 5.3.1 and 5.3.2 are both satisfied).*

It follows that given two rule transformations, *y_t1 = rule1(x, x_t, op)* and *y_t2= rule2(x, x_t, op)*, that do not have any false-negatives, *rule1* over-taints *rule2* if there is a bit in y_t1 that is 1, but 0 in y_t2 (Equation 5.6).

Definition 5.3.4. *Given two accurate rules* $y\_t2 = rule1(x, x\_t, op)$ *and* $y\_t2 = rule2(x, x\_t, op)$, rule1 *over-taints* rule2 *if the following condition is true.*

$$(y\_t1\ \&\ (y\_t1 \oplus y\_t2)) \neq 0 \tag{5.6}$$

Note that these rules are initial state agnostic. They can be changed to support analyzing the flows between data items (i.e., a subset of the initial states), x_i and y_j by

assuming that only the bits in the state that corresponds to x_i can be tainted. All other bits are not tainted.

5.4 Generating an Accurate Policy for x86

In this section, a case study is used to illustrate how the foundation set in the previous sections can be applied. The case study involves first using the definitions to generate an accurate policy for x86 and then comparing it to previously published taint trackers with regards to accuracy and precision. While the focus is on x86, the results are applicable to all bitvector based implementations. The methodology for generating the policy is presented in this section and interpretation of the results is left to the next.

The implication, $s \rhd_0 r \implies (s, r) \in T$, is used to ensure that the generated policy is accurate. That is, the policy is defined by analyzing the information flows. As discussed in Section 5.2.3, applying this implication ensures that every $(s, r) \in T_G$ must also $\in T$ and thus the policy T does not contain any false-negatives (Definition 5.2.4).

A two-step process is used to identify the information flows in common x86 instructions. First, the behavior of each instruction is specified using bitvector semantics. The output of this stage is a collection of SMT-LIB Version 2 [134] or SMT2 files. Second, the condition in Definition 5.2.1 is tested on the behavioral definitions to obtain a general portrait of the bit-wise information flow relationships between the instruction operands. The output of this stage is a collection of directed graphs depicting the bit-wise information flows, which are subsequently interpreted manually to obtain the conservative accurate taint propagation policy.

5.4.1 Stage 1: Behavioral Definitions

Similar to previously published information flow trackers, the case study target the general purpose x86 instructions that fall into the *data transfer*, *arithmetic* and *logical* categories. *Data transfer* instructions have simple semantics with obvious bit-wise information flow relationships. Thus, only the *arithmetic* and *logical* instructions are studied. Then, in an effort to reduce redundancy, the analysis is limited to the 32 bit instruction formats. The different mnemonics (e.g., `add eax,imm32` vs. `add r/m32, imm32` vs. `add r32, r/m32`) where the only difference between them are the operand types, are ignored. The `bsf`, `bsr` and `cmpxchg` instructions are included since they include conditional behavior and will be used to illustrate over-tainting when propagating taint through an IR.

The first step is to determine all of the input and output parameters, including flags by referring to the developer's manuals. This is done for each instruction. These parameters constitute the state S. Only flags whose behaviors are well defined are included. Flags that are unchanged or undefined are assumed not to exhibit information flow and are excluded from the state.

Since the accuracy of the behavioral definitions is paramount, the definitions are cross referenced with both BAP [135] and the developer's manuals. Given an instruction, assembly code to exercise different aspects of the it are written and compiled into an executable. BAP 0.4 was then used to translate the executable into BAP's internal IR. The generalized behavior, expressed in SMT2, was then extrapolated from the IR for different instruction instantiations as well as the manuals. Godefroid and Taly [136] presented

```
(define-fun f_ror_32 ((S STATE)) (_ BitVec 32)
  (bvor
    ((_ extract 31 0) (bvlshr (concat (rm32 S) #x00000000)
           (concat #x00000000 (f_shiftcount S))))
    ((_ extract 63 32) (bvlshr (concat (rm32 S) #x00000000)
           (concat #x00000000 (f_shiftcount S))))
  )
)
(declare-const f_cf_const (_ BitVec 1))
(define-fun f_cf ((S STATE)) (_ BitVec 1)
  (ite (= (f_shiftcount S) #x00000000)
    f_cf_const ; undefined or unchanged
    ((_ extract 31 31) (f_ror_32 S))
  )
)
(declare-const f_of_const (_ BitVec 1))
(define-fun f_of ((S STATE)) (_ BitVec 1)
  (ite (= (f_shiftcount S) #x00000001)
    (bvxor ((_ extract 31 31) (f_ror_32 S))
           ((_ extract 30 30) (f_ror_32 S)) )
    f_of_const; undefined or unchanged
  )
)
```

Fig. 5.1.: SMT2 Definition for `ror dst, imm8`

```
T_32t0_250:u32} = R_EAX:u32
T_32t6_256:u32 = T_32t0_250:u32 << pad:u32(0x1e:u8)
T_32t8_258:u32 = T_32t0_250:u32 >> pad:u32(2:u8)
T_32t5_255:u32 = T_32t8_258:u32 | T_32t6_256:u32
R_CF:bool = low:bool(T_32t5_255:u32 >> 0x1f:u32)
R_OF:bool = R_CF:bool ^ low:bool(T_32t5_255:u32 >> 0x1e:u32)
```

Fig. 5.2.: BAP IL for `ror %eax, $0x2`

algorithms to automatically generate these behavioral specifications. Implementation of the algorithms is left as future work.

The `ror dst, imm8` instruction is used as an example. According to the manual, `ror` has two operands, `dst` (the 32-bit operand to right rotate) and `imm8` (the number of bit positions to rotate). Since `ror` affects the overflow and carry flags, S is 42 bits in length. Figure 5.1 shows the SMT2 definitions for the `ror %eax, $0x2` instruction and Figure 5.2 shows the corresponding BAP IL; output for other instantiations are not shown. Note that `f_ror_32` uses an alternative logic but the flags calculations are based on BAP and the documentation.

```
(define-sort STATE () (_ BitVec 70))
(define-fun dst ((S STATE)) (_ BitVec 32)
  ((_ extract 69 38) S)
)
(define-fun f_add ((S STATE)) (_ BitVec 32)
  (bvadd (dst S) (src S))
)
(define-fun f_of ((S STATE)) (_ BitVec 1)
  ((_ extract 31 31)
    (bvand ( bvxor (dst S) (src S) #xFFFFFFFF )
           ( bvxor (dst S) (f_add S) )
) ) )

;;;Other function definitions

(declare-const DST (_ BitVec 32))
(declare-const SRC (_ BitVec 32))
(declare-const ZF  (_ BitVec  1))
;;;Other declarations

(assert (exists ( (i (_ BitVec 32)) (j (_ BitVec 32)) )
          (not (= (dst (add (concat DST i ZF OF SF AF CF PF) ) )
                  (dst (add (concat DST j ZF OF SF AF CF PF) ) )
)       ) )    )
(check-sat)
```

Fig. 5.3.: SMT2 Definition and Test for `add dst, src`

As another quick example, a portion of the SMT2 definition for `add` is shown in Figure 5.3. The figure shows that `add` has an SMT2 equivalent operation `bvadd` meaning it is a basic operation. The flags calculation logic was extracted from BAP.

5.4.2 Stage 2: General Information Flow

The goal of this stage is to take the SMT2 files from stage 1 and obtain a base set of taint propagation policies. For each file and for each possible combination of input and output bits of the state S, Z3 3.2 was queried for the satisfiability of the condition in Definition 5.2.1. An example query can be found at the bottom of Figure 5.3. The query is used to determine whether there are two values i and j of `src` such that the values of `dst` are different after `add`. The bit-wise query is more involved, however, it follows the same

pattern. For example, `ror` has `dst` and `imm8` as inputs totaling 40 bits and `dst`, `imm8`, `of`, `cf` as outputs, totaling 42 bits. This results in 1680 separate queries.

The resulting statistics for all the instructions are summarized in the first five columns of Table 5.1. The instructions are presented in the first column, the input operands, both implicit and explicit, in the second, output operands, both implicit and explicit, in the third, the total number of input-bit to output-bit combinations in column four and the time it took for Z3 to return "sat" or "unsat" results for the condition in Definition 5.2.1 is shown in column five. A new instance of Z3 is used for each test case and thus the results include process creation overhead.

As expected, logical operations return results extremely quickly whereas signed multiply and divide takes the most time. Overall, it took less than 14 hours on an Intel Core-i7 860 to automatically identify all information flow relationships for 26 arithmetic and logical instructions. However, the multiply and divide operations, especially the signed versions, took the longest time by far. This indicates that the same technique will have limited use for FPU and MMX instructions unless theorem provers improve. Further analysis is left as future work.

5.5 Results

This results section is separated into four parts that illustrate the different aspects of the formal foundations laid out in Sections 5.2 and 5.3: 1) the sat/unsat results generated from stage 2, as described in the previous section, are interpreted; 2) previously published taint analysis platforms are compared against the automatically generated policy; 3) the

						Previously Published Taint Trackers								
Instruction	**Inputs**	**Outputs**	**# Cases**	**Runtime**	**Flow Type**	**DroidScope**	**Cat1**	**libdft [95]**	**Minemu [103]**	**Cat2**	**TEMU [16]**	**Cat3**	**Memcheck [84]**	**Refined**
adc *dst, src*	dst,src,cf	dsr,src,zf,of,sf,af,cf,pf	4550	1m19s	U	**A**	**A**	I	**A**	**A**	S	S	U	S
add *dst, src*	dst,src	dst,src,zf,of,sf,af,cf,pf	4480	1m13s	U	**A**	**A**	I	**A**	**A**	**A**	**A**	S	S
and *dst, src*	dst,src	dst,src,zf,sf,pf	4288	1m05s	I	**A**	I	I	**A**	I	I	S	S	S
dec *dst*	dst	dst,zf,of,sf,af,pf	1184	20s	U	**A**	**A**	I	**A**	**A**	**A**	**A**	U	U
div *rm32*	edx,eax,rm32	edx,eax,rm32	9216	95m48s	D	**A**	**A**	I	N	**A**	**A**	**A**	**A**	D
idiv *rm32*	edx,eax,rm32	edx,eax,rm32	9216	307m04	A	A	A	I	N	A	A	A	A	A
imul1 *rm32*	eax,rm32	edx,eax,rm32,of,cf	6272	289m51s	U	**A**	**A**	I	N	**A**	**A**	**A**	U	U
imul2 *dst, rm32*	dst,rm32	dst,rm32,of,cf	4224	52m37s	U	**A**	**A**	I	N	**A**	**A**	**A**	U	U
imul3 *dst, rm32, imm32*	rm32,imm32	dst,rm32,imm32,of,cf	6272	53m56s	U	**A**	**A**	I	N	**A**	**A**	**A**	U	U
inc *dst*	dst	dst,zf,of,sf,af,pf	1184	19s	U	**A**	**A**	I	**A**	**A**	**A**	**A**	U	U
mul *rm32*	eax,rm32	edx,eax,rm32,of,cf	6272	16m02s	U	**A**	**A**	I	N	**A**	**A**	**A**	U	U
not *dst*	dst	dst	1024	15s	I	**A**	I	I	**A**	I	I	I	I	I
or *dst, src*	dst,src	dst,src,zf,sf,pf	4288	1m05s	I	**A**	I	I	**A**	I	I	S	S	S
rcl *dst, imm8*	dst,imm8,cf	dst,imm8,of,cf	1722	42s	A	A	A	N	A	A	A	A	A	S
rcr *dst, imm8*	dst,imm8,cf	dst,imm8,of,cf	1722	42s	A	A	A	N	A	A	A	A	A	S
rol *dst, imm8*	dst,imm8	dst,imm8,of,cf	1680	41s	A	A	A	N	A	A	A	A	S	S
ror *dst, imm8*	dst,imm8	dst,imm8,of,cf	1680	41s	A	A	A	N	A	A	A	A	S	S
sal *dst, imm8*	dst,imm8	dst,imm8,zf,of,sf,af,cf,pf	1840	35s	U	**A**	**A**	N	**A**	**A**	S	S	S	S
sar *dst, imm8*	dst,imm8	dst,imm8,zf,of,sf,af,cf,pf	1840	34s	D	**A**	**A**	N	**A**	**A**	S	S	S	S
sbb *dst, src*	dst,src,cf	dst,src,zf,of,sf,af,cf,pf	4550	1m21s	U	**A**	**A**	I*	**A***	**A**	**A**	**A***	**A**	S
shr *dst, imm8*	dst,imm8	dst,imm8,zf,of,sf,af,cf,pf	1840	35s	D	**A**	**A**	N	**A**	**A**	S	S	S	S
sub *dst, src*	dst,src	dst,src,zf,of,sf,af,cf,pf	4480	1m17s	U	**A**	**A**	I*	**A***	**A***	**A***	**A***	S	S
xor *dst, src*	dst,src	dsr,src,zf,sf,pf	4288	1m05s	I	**A**	I	I*	**A***	**A***	**A***	**A***	I	I
bsf *dst, src*	src	dst,src,zf	2080	31s	A	N	A	I	N	A	A	A	A	S
bsr *dst, src*	src	dst,src,zf	2080	31s	S	N	**A**	I	N	**A**	**A**	**A**	**A**	S
cmpxchg *rm32, r32*	eax,rm32,r32	eax,rm32,r32,zf,of,sf,af,cf,pf	9792	2m39s	S	N	**E**	**E**	N	**E**	**E**	**E**	**E**	S
TOTAL			102064	13h52m48s										

Table 5.1: Flow Type Results for x86 Instructions

Flow Types: (U)p, (D)own, (I)n-place, (A)ll-around, (S)pecial, (N)ot-Supported, (S)pecial, (E)ax is tainted in `cmpxchg`,
* - Zeroing Idiom, **Boldface** - Generated Policy is more precise

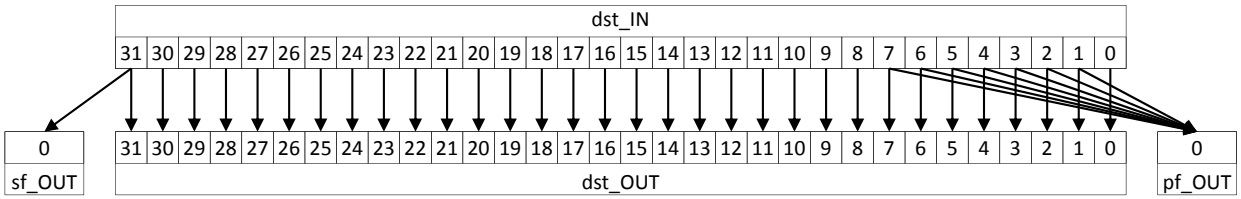

Fig. 5.4.: Information flows of `dst` in the `or` instruction

special rules in Memcheck are verified; and 4) previously published taint trackers are compared based on taint- and analysis-granularity.

5.5.1 Interpretation of Results

The sixth column of Table 5.1 indicates the general flow type for each instruction. The sat/unsat results from Z3 were first graphed to increase understandability, and then interpreted to determine the flow types. In particular, one directed graph was generated per input bit. The nodes are bits of the state S and edges signify the potential for information flow from the source bit to the destination bit. As an example, the 32 graphs from the 32 input bits of `dst` in `or` were combined to produce Figure 5.4. The bit to bit *in-place* information flow relationship is evident.

Information Flow Types

There are four distinct information flow patterns between the source and destination operands. Despite the fact that information flow was analyzed at the bit-level, the patterns are the same for a byte-level taint-granularity. There are no patterns of interest for the flags. In other words, the results match the definitions and are therefore expected.

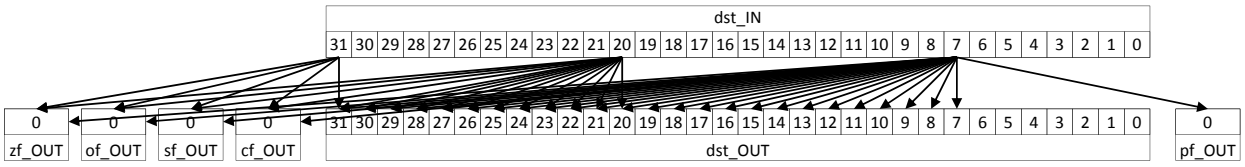

Fig. 5.5.: Information flow of bits 7, 20 and 31 of `dst` in `sbb`

1. *In-place:* Information can only flow from bit i of the source to bit i of the destination. The Memcheck equivalent is *UifU*.

2. *Up:* Information can only flow from bit i of the source to bits j of the destination where $j >= i$. Figure 5.5 depicts this particular behavior. It is the combination of the information flow graphs for bits 7, 20, and 31 of `dst` to `dst`. It is evident from the figure that information only flows from bit 7 of the source operand to bit 7 and higher of the destination. The same applies to bits 20 and 31, where bit 31 of the source only flows to bit 31 of the destination. The Memcheck equivalent is *Left*.

3. *Down:* Information can only flow from bit i of the source to bits j of the destination where $j <= i$. While Memcheck does not have an equivalent operation, it is trivial to see that, if there was, it would be *Right*.

4. *All-around:* Information can flow from bit i of the source to any bit of the destination. The Memcheck equivalent is *Lazy*.

It is possible for an instruction to exhibit multiple flow types at the same time. The flow types listed in Table 5.1 only shows the most descriptive type. The divide instructions are good examples of this. In divide, `edx:eax` is divided by `rm32`, the quotient placed into `eax` and remainder into `edx`. Intuitively, division is similar to shift right and thus the flow

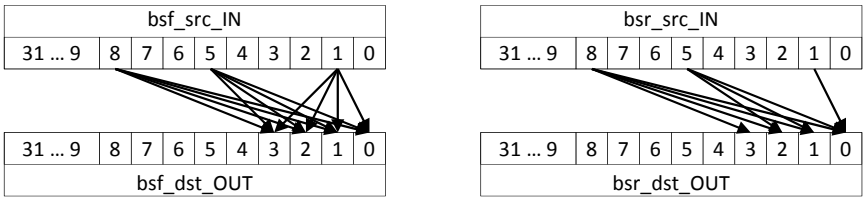

Fig. 5.6.: Comparison between `bsf` and `bsr`

type for `edx:eax` to `eax` should be *down*. On the other hand, the flow type for `edx:eax` to `edx` is *all-around* since nothing definitive can be said about the relationship between the divisor and the remainder without conrete value analysis. Also in practice, the actual taint propagation policy will either be the *OR* of the individual flow types between the input operands and the output operands or special rules defined per case. All in all, the flow types identified in this step are conservative and are used to define the accurate taint propagation policy.

The `bsf`, `bsr` and `cmpxchg` Instructions

`bsf` and `bsr` are two instructions that iterate through bit positions to find the first 1-bit. If these instructions do not have native IR counterparts, then it must be emulated. Since the software developer's manuals describe the instructions using while loops, they both have control flow behaviors. The resulting flows for input bits 1, 5 and 8 are depicted in Figure 5.6. Since `bsf` tests the lowest bits first, all subsequent iterations of the loop depend on the lower bits being '0' and thus the flow behavior is equivalent to all-around. For example, if only bit number 5 of `src_IN` is 1, then bits 0,1 and 2 of `dst_OUT` are set. However, if bit number 1 is also set, then only bit 0 of `dst_OUT` is set. Thus, the setting of bits 1 and 2 depend on the fact that bit 1 of `src_IN` is 0. `bsr` scans from the highest bit

```
1. cmpxchg (rm32, r32) {
2.   if (eax == rm32 ) then
3.     rm32 = r32;
4.   else eax = rm32;
5. }
```

Fig. 5.7.: Pseudocode for `cmpxchg` (flags are omitted)

position down, thus the flow pattern is much more direct and is special, not all-around. These complex behaviors can also be identified as control flow dependencies using the technique proposed by Ferrante et al. [137]. Both Dytan and DTA++ uses this technique; however, control flow dependency does not immediately reveal the special bit-level relationships of `bsr` as shown in Figure 5.6. Consequently, their policies are classified as all-around. All other policies are assumed to support these instructions using all-around unless evidence to the contrary was found.

`cmpxchg` (Figure 5.7) is a special x86 instruction that can be used to show how IR tainting can lead to false-positives. Applying Definition 5.2.1 on the instruction as a whole shows that there is no information flow from `eax` to `eax` because the output value of `eax` is fully dependent on the input value of `rm32`. On the other hand, if information flow was analyzed line-by-line, and thus mimicking the behavior of IR based tainting where each line is a corresponding IR, `eax` will be tainted if `eax` was tainted before the instruction. This is because `eax` was unchanged in the equals branch (line 3) and thus retains its taint. The case for simple control flow dependencies is even worse. Since `eax` is used in the comparison on line 2 and also as an l-val on line 4, it will remain tainted in the not-equals branch. In the end, care must be taken to ensure that IR level tainting does not introduce false-positives or false-negatives.

5.5.2 Comparing With Previously Published Policies

In practice, many taint propagation policies are manually defined using domain knowledge and expertise. This naturally leads to the questions of whether these manually defined policies are accurate and if so how precise, and is automatic synthesis necessary? These questions can be answered using the definitions presented in Section 5.2.3 and the accurate policy generated in the previous section.

While the automatically generated policy was defined for the x86 instruction set, the bitvector semantics are common to other platforms as well. However, not all taint analysis implementations use the same bitvector operations and therefore the comparison is limited to those that do.

Bit-vector based policies are separated into three broad categories. Cat1 includes all policies that rely on simple l-val r-val relationships [88, 94, 103, 105, 108, 111]. Effectively, these policies consists of *in-place* and *all-around* flow types only. There are three special cases in this category that are discuss in more detail.

First, RIFLE [105] is a hardware implementation that associates taints with operands and relies on the *join* operation, which is essentially an *OR*. Since taint is assigned per operand, the propagate *up* and *down* flow types are meaningless. Thus, they are simplified into *all-around* to maintain accuracy.

Then, the published policy for arithmetic and logical operations in GLIFT [108] is equivalent to *all-around*. This is strange since the paper used an *AND* gate to illustrate the special case where if the untainted input is 0 then the output is always 0, and thus taint at

the output should be cleared. Yet, the final policy presented does not handle this special case. For consistency, GLIFT was placed into Cat1. It is likely Cat3 though.

Finally, in Raksha [111], Dalton et al. limited the taint propagation policies to three operations, no propagation, *AND* the taint values and *OR* the taint values. These operations are then defined for instruction classes (e.g., arithmetic), instead of individual instructions (e.g., `add` and `sub`). Raksha does provide support for defining special rules for up to four operations though. Since the comparison is between base policies, Raksha is categorized into Cat1, although in practice is is most likely between Cat1 and Cat2. Similarly, Dytan [94] and Minemu [103] allows users to define they own propagation policies as well.

Cat2 and Cat3 include all policies with special cases support. Cat2 includes simple special cases due to zeroing idioms like `xor eax, eax` [59, 95–97, 106, 107] and Cat3 includes more precise policies that handle many special cases including the shift operations [16, 56, 102, 109, 110].

The results are shown on the right side of Table 5.1. Since taint propagation policies can be updated after publication, all of the taint propagation implementations that has source code available has their own column with the policy defined in the source. The policies are ordered by precision with the least precise on the left (column 7) and the most precise on the right (column 14).

As the results show, many of the default policies used in previous publications are not as precise as the automatically generated policy. In fact, none of the policies used the propagate up flow type of `sbb`. Another interesting point is that the default policies for libdft and Minemu are not accurate (i.e., has false-negatives). Either some of the

instructions were not supported, or in the case of libdft, the default use of union or *OR* leads to false-negatives in instructions such as `add` where taint should propagate up. Finally, the results indicate that Memcheck has the most special rules and therefore could be the most precise taint analyzer surveyed. The accuracy and precision of Memcheck's rules are verified in the next section.

5.5.3 Refining Memcheck's Special Rules

In order to verify Memcheck's rules, Memcheck's operations (e.g., *UifU*) were first defined in SMT2 and then used to define rules from both the original paper [84] as well as the Memcheck source code. These taint transformation rules are then analyzed for false-negatives and false-positives based on Equations 5.4 and 5.5 respectively. Furthermore, to maintain consistency, the rest of the discussion is framed using Memcheck's operations and nomenclature. In particular, data operands are labeled using d* where * is the operand number (e.g., d1 for the first operand), and the shadow taint variables are labeled as v* where * is the operand number of the operand it shadows (e.g., v1 is the shadow taint variable for d1).

As an example, Figure 5.8 shows the logic for accurately calculating the resulting taint *v3* for the *AND(d1, d2)* instruction with 32bit operands as declared in SMT2. The original definition from the Memcheck [84] is shown in comments (i.e., lines that start with ';'). To determine whether the rule is accurate, a simple query to the SAT solver is used to determine whether Equation 5.5 holds for all input values of d1, v1, d2 and v2. This is

```
;Declaration for AND
(define-fun OP ( (d1 (_ BitVec 32)) (d2 (_ BitVec 32)) )
               (_ BitVec 32)
  (bvand d1 d2)
)

;Original Memcheck logic
;v3 = DifD( UifU(v1,v2),
;           DifD( ImproveAND(d1,v1),
;                 ImproveAND(d2,v2) ) )
(define-fun rule ( (d1 (_ BitVec 32)) (d2 (_ BitVec 32))
                   (v1 (_ BitVec 32)) (v2 (_ BitVec 32)) )
                 (_ BitVec 32)
  (DifD (UifU v1 v2)
        (DifD (ImproveAND d1 v1)
              (ImproveAND d2 v2)
        )
  )
)
```

Fig. 5.8.: Comparison between Memcheck logic and SMT2 code for verifying AND

achieved by asserting that it is not true, such that an unsat result will signify that it must hold.

The results are summarized in Table 5.2. The `xor` instruction was included as a test of the Z3 theorem prover. Memcheck does not have any special rules for `xor`. Runtime numbers are not included since the false-negative tests all returned quickly, within a few minutes, and most of the false-positive tests timed out (i.e., Z3 did not return sat or unsat within 24 hours). The instruction, SAT solver result for the false-negative test and the result for the false-positive test are presented in that order. The table is also separated into two sections, the first contains the rules defined in Memcheck and the second contains additional refined rules.

The results indicate that all of the Memcheck rules are accurate and in fact two of them, `and` and `cmp`, are precise. Furthermore, the initial results for the false positive tests for the shift and rotate instructions returned models (i.e., variable assignments that satisfies the query) highlighting the fact that the result is tainted if the shift amount is

Instruction	False-Negatives	False-Positives
add	unsat	timeout
and	unsat	unsat
cmp	unsat	unsat
or	unsat	timeout
rol	unsat	sat, timeout*
ror	unsat	sat, timeout*
sal	unsat	sat, timeout*
sar	unsat	sat, timeout*
shr	unsat	sat, timeout*
sub	unsat	timeout
xor	unsat	timeout
New Rules		
adc	unsat	timeout
rcl	unsat	sat, timeout*
rcr	unsat	sat, timeout*
sbb	unsat	timeout
bsf	unsat	sat*, unsat
bsr	unsat	sat*, unsat

Table 5.2: Summary of refined policies

tainted. Since this result is of limited use, the natural question is whether the only source of false-positives is from the shift amount. Hence, the search space was limited so that the shift amount is never tainted. This new test resulted in a timeout, noted by the *.

Despite the fact that Z3 timed out for some false-positive tests, conclusions about a rule's precision are not drawn. This is mainly because when the same queries were issued in Z3 4.0, "unknown" was returned. According to the Z3 documentation this is likely due to the use of quantifiers, which Z3 is not optimized for. This is, at least for now, a limitation of the proposed approach. On a more promising note, Z3 returned unsat in about 40 minutes for 4-bit `xor` and in about 664 minutes for 8-bit `xor`. It might be possible to manually extend these lower-bit count results to larger operands. This is left as future work.

There are similarities between the addition and shift operations, and the addition with carry and rotate operations. Thus, it was natural to attempt to define new rules for the latter operations based on the prior and verify their accuracy and precision. The list of new rules are presented in Table 5.3 with the instruction on the first column and a description of the new rule in the second. The rules for `rcl` and `rcr` are natural extensions of the shift and rotate rules identified in Memcheck. The taint value for the carry flag, *vcf*, is included in the calculations. UifU is a function that bitwise ORs the operands and PCastYX, the pessimistic cast, returns 0 if the operand is 0 and 1s in all bit positions otherwise.

The rules for `adc` and `sbb` are also natural extensions of the `add` and `sub` rules. The *min* terms represent the value where all tainted bits are set to 0 (e.g., d1_min = d1 & ˜ v1), and the *max* terms represent the value where all tainted bits are set to 1 (e.g., d1_max = d1 | v1).

Insn	Refined Rule
adc	(bvor (bvor v1 v2) (bvxor (bvadd d1_min d2_min cf_min) (bvadd d1_max d2_max cf_max)))
sbb	(bvor (bvor v1 v2) (bvxor (bvsub d1_min (bvadd d2_max cf_max)) (bvsub d1_max (bvadd d2_min cf_min))))
rcl	(UifU (PCastYX v2) (rcl v1 d2 vcf)) where vcf is the taint value for the carry flag
rcr	(UifU (PCastYX v2) (rcr v1 d2 vcf)) where vcf is the taint value for the carry flag
bsf	Destination is tainted if (bvule (bsf v1) (bsf d1))
bsr	Destination is tainted if (bvuge (bsr v1) (bsr d1))

Table 5.3: Summary of new rules using SMT2 prefix notation

The `bsf` and `bsr` rules are similar to the Memcheck rule for `cmp` where short-circuiting is used. The intuition is that if an untainted bit is already 1, then it doesn't matter what the value of the tainted bit is as long as the tainted bit is scanned after the untainted bit.

5.5.4 Taint- and Analysis-granularity

Section 5.2.2 discussed how taint- and analysis-granularity may affect taint propagation precision. As an illustration, those definitions are used to compare the precision of previously published taint trackers.

Taint-granularity Much research has been conducted to propagate taint through bitvector operations. Policies have been defined at the operand level [87, 105], 32-bit word level [59, 111], byte-level [56, 59, 88, 95–97, 106, 107, 109, 110], and bit-level [84, 108]. According to Theorem 5.3.1 the taint-granularities have been ordered with increasing precision.

Other taint analysis applications, such as the ones on Java String objects, can be compared analogously as well. Like how bitvectors consists of bits or bytes, Java Strings

consists of UTF-16 characters or full String objects. Therefore, theorem 5.3.1 indicates that the Java String propagation policies defined at the character level [91, 92] may be more precise than those defined at the object level [93, 99]

Analysis-granularity: The range of analysis-granularity designs matches well with the different levels of programming languages. Researchers have defined propagation policies for high-level languages such as C [87] and Java [91–93, 99], scripting languages such as PHP [89], PERL [101] and JavaScript [90, 100], low level languages such as x86 assembly [56, 84, 94, 103, 104] and even at the gate level [108].

It is also common practice to implement taint propagation through an intermediate language [56, 59, 84, 94, 104] with simpler semantics and reduced instruction set than through the language or instruction set (e.g., x86 and ARM), they emulate. The emulation code can range from a single instruction, to a basic block of instructions to whole functions that contain multiple execution paths. Consequently, Theorem 5.3.2 indicates that this design choice may increase false-positives and reduce precision. Similar to Memcheck, the presence of false-positives and negatives can be verified.

5.5.5 ARM and Dalvik Level Tainting in DroidScope

Similar to how DroidScope limited fine-grained analysis support to Dalvik bytecodes emulated using mterp, the discussions on the precision of analyzing Dalvik level taint using only ARM instructions is also limited to the mterp interpreter. As discussed in Section 4.4.2, each bytecode in mterp is associated with a set of native instructions that is used to emulate the bytecode's functionality. Since the native code is optimized for individual

```
 1. %default {"preinstr":"", "result":"r0", "chkzero":"0"}
 2. /*
 3.  * Generic 32-bit binary operation.  Provide an "instr" line that
 4.  * specifies an instruction that performs "result = r0 op r1".
 5.  * This could be an ARM instruction or a function call.  (If the result
 6.  * comes back in a register other than r0, you can override "result".)
 7.  *
 8.  * If "chkzero" is set to 1, we perform a divide-by-zero check on
 9.  * vCC (r1).  Useful for integer division and modulus.  Note that we
10.  * *don't* check for (INT_MIN / -1) here, because the ARM math lib
11.  * handles it correctly.
12.  *
13.  * For: add-int, sub-int, mul-int, div-int, rem-int, and-int, or-int,
14.  *      xor-int, shl-int, shr-int, ushr-int, add-float, sub-float,
15.  *      mul-float, div-float, rem-float
16.  */
17. /* binop vAA, vBB, vCC */
18. FETCH(r0, 1)                        @ r0<- CCBB
19. mov     r9, rINST, lsr #8           @ r9<- AA
20. mov     r3, r0, lsr #8              @ r3<- CC
21. and     r2, r0, #255                @ r2<- BB
22. GET_VREG(r1, r3)                    @ r1<- vCC
23. GET_VREG(r0, r2)                    @ r0<- vBB
24. .if $chkzero
25. cmp     r1, #0                      @ is second operand zero?
26. beq     common_errDivideByZero
27. .endif
28.
29. FETCH_ADVANCE_INST(2)               @ advance rPC, load rINST
30. $preinstr                           @ optional op; may set condition codes
31. $instr                              @ $result<- op, r0-r3 changed
32. GET_INST_OPCODE(ip)                 @ extract opcode from rINST
33. SET_VREG($result, r9)               @ vAA<- $result
34. GOTO_OPCODE(ip)                     @ jump to next instruction
```

Fig. 5.9.: ARM Emulation code for basic mterp operations (binop.S)

architectures, the discussions are focused on ARM v5 architecture, which is widely supported.

In the ARM v5 implementation of mterp, all of the arithmetic operations are generated using macros. The basic emulation code for 32-bit binary arithmetic opcodes is defined in a file named `binop.S` and is depicted in Figure 5.9. As the figure shows, mterp first loads the Dalvik virtual register numbers into scratch registers `r2` and `r3` (lines 20 and 21), then loads the operands from the stack using the `GET_VREG` macro into scratch registers `r0` and `r1` (lines 22 and 23), operates on the registers using the `$instr` macro (line 31), and finally loads the result back into the corresponding virtual register (line 33). For example, the

`ADD_INT` mterp opcode effectively replaces line 31 with `add r0, r0, r1` and `DIV_INT` replaces line 31 with `bl __aeabi_idiv`, the call to the native function that performs division since some ARM architectures, v5 included, do not support division natively.

Since the only non-data-transfer instruction in the emulation code is the actual arithmetic operation (e.g., add), the taint propagation policy for tracking taint at the ARM instruction level is the same as the logic for tracking taint at the Dalvik opcode level. The lines of interest are 22, 23, 31 and 33 and in fact are no different than a basic block of code that loads memory operands into registers for processing and then stores the result from the register back into memory.

The only caveat is that the Dalvik opcodes are assumed not to be tainted. If they are tainted, then the instructions used to decode and interpret the bytecode (lines 18-21) will blindly propagate the taint through. This might or might not be the desired behavior since these details do not exist in Dalvik level taint propagation. The advantage of propagating taint through ARM instructions is the added control over how taint is to be propagated through the decoding process. On the other hand, the `and` and `lsr` (lines 19-21) need to be precise in order to avoid over-tainting. Special rules for these operations were discussed in Section 5.4 and are likely to be precise.

Unary operations are emulated similarly and thus for all practical purposes, propagating taint at the native instruction level is equivalent to propagating taint at the Dalvik opcode level for the arithmetic instructions.

5.6 Discussion

SMT Solvers The ability to show accuracy and precision is wholly dependent on SAT solvers such as Z3. As the timing results show, even queries to determine whether simple rules, such as `xor`, is precise do not return within 24 hours. This is mainly due to the use of quantifiers in the definitions and equations, which Z3 is not optimized for. On the other hand though, it should be stressed that the proofs for accuracy return quickly and thus, in the minimum, it is feasible to ensure that policies do not have false-negatives. Other solvers can be used to solve the same queries. This is left as future work.

Other Design Considerations The comparisons have focused on the design parameters that can be effectively measured with the proposed approach. There are other extremely important design constraints such as performance and generality. That is, while the refined policies decrease false-positives, and therefore the over-tainting problem, the overhead needed for applying the rules can greatly affect system performance. Similarly, while a finer-grained taint-granularity increases precision, it also requires more resources to maintain the additional taint tags.

Overall, the final decision on how to balance the trade-offs between accuracy, precision, performance and other factors is application dependent. There are already indications that the trade-offs mentioned above can be successfully managed though. This means that precise policies can be used in practice.

Symmetric multi-threading [107], unused extension processing units, (e.g. FPU/MMU) [103], special hardware [111] and mixed static and dynamic analysis [138] have been and can still be used to improve the performance of calculating taints. This is

especially true for the complex refined policies. To reduce taint label storage requirements, researchers have used mixed granularities [59, 96] and complex page table like structures [87, 95].

There is also precedence in pairing taint analysis with SMT solvers at runtime [9, 139], mixing analysis-granularities by using function summaries [56, 96], adjusting analysis-granularity at runtime [104] and dynamically selecting what to analyze [40, 97]. Additionally, the concept of Secure Multi-Execution [140] has been shown capable of determining whether a program is "noninterferent" with little overhead in the JavaScript engines of Chrome and Firefox [141]. These previous efforts indicate that calculating precise information flows at runtime might be feasible, even for larger analysis-granularities such as functions.

Automatic Synthesis Both the behavioral models and the refined special rules were manually defined, which can still introduce errors. To achieve the highest precision, the policies as well as the special rules should be completely automatically generated. To this extent, Godefroid and Taly presented a "Smart" algorithm that uses *function templates* to automatically synthesize the behavioral models of the same x86 instructions (excluding `adc` and `sbb`) used in the tests [136]. The same algorithm can be used to synthesize the refined rules as well as the special sanitization rules. It can be seen that the rules for `adc`, `sbb`, `add` and `sub` follow a similar template and so do the sanitization rules. This problem is left as future work.

5.7 Conclusion

The purpose of this chapter was to develop the fundamental understanding and methodology for analyzing the accuracy and precision of dynamic taint propagation policies. This was achieved by deriving the noninterfering data model and using it to make four main observations on the fundamental sources of false-positives and false-negatives in dynamic taint analysis implementations. It was proven that byte level tainting strictly over-taints bit level tainting and that basic block level tainting, and in essence IR level tainting, strictly over-taints instruction level tainting. The model was also used to formally define previously used terms and to define definitions for verifying accuracy and precision.

The merit of the more formal approach was shown through a case study for generating precise taint propagation policies for common bitvector operations. The automatically identify information flow based policy combined with the refined rules were verified to be more precise than previously published taint trackers. The implementation of a new more precise taint tracker is left as future work.

6. SUMMARY

The original thesis was that transparent malware analysis platforms with precise taint tracking rules can be realized using virtualization technologies. This thesis has been validated in this dissertation.

V2E showed that heterogeneous record and replay can be used to record a sample's execution in a hardward virtualized environment that is *transparent*, followed by a replay in an emulated environment. By changing the emulator to precisely replay the log as recorded, the emulation based malware analysis platform, TEMU, was able to successfully analyze real-world malware samples that previously evaded its analysis.

DroidScope showed that the two levels of semantic gaps in Android malware can be bridged. Furthermore, it showed that two levels of semantic information can also be seamlessly bound together such that a single emulation based malware analysis platform can be used to analyze Android malware with colluding Java and native components.

Both V2E and DroidScope are emulation based malware analysis platforms and thus benefit from the advantages of *flexibility* and *efficiency* for analysis plugin development. They both have instruction tracer plugins and support taint analysis as well.

The accuracy and precision of the taint propagation policies in both of these implementations as well as other previously published taint analysis platforms were analyzed and compared using the methods presented in Chapter 5. The results showed room for improvement.

In conclusion, while transparent and precise malware analysis is feasible using virtualization, future work (as mentioned throughout this dissertation) is needed to extend these initial results.

LIST OF REFERENCES

LIST OF REFERENCES

[1] "Online exposure social networks, mobile phones, and scams can threaten your security," *Consumer Reports*, vol. 76, June 2011.

[2] "Norton study calculates cost of global cybercrime: $114 billion annually." `http://www.symantec.com/about/news/release/article.jsp?prid=20110907_02`, 2011.

[3] Z. Bu, T. Dirro, P. Greve, D. Marcus, F. Paget, R. Permeh, V. Pogulievsky, C. Schmugar, J. Shah, P. Szor, and A. Wosotowsky, "Mcafee threats report: Fourth quarter 2011," tech. rep., McAfee Labs, 2012.

[4] Z. Bu, T. Dirro, P. Greve, D. Marcus, F. Paget, R. Permeh, C. Schmugar, J. Shah, P. Szor, G. Venere, and A. Wosotowsky, "2012 threats predictions," tech. rep., McAfee Labs, 2011.

[5] "State of mobile security 2012," tech. rep., lookout Mobile Security, 2012.

[6] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Comput. Surv.*, vol. 44, pp. 6:1–6:42, Mar. 2008.

[7] F. Bellard, "QEMU, a fast and portable dynamic translator," in *USENIX Annual Technical Conference, FREENIX Track*, April 2005.

[8] "Qemu." `http://www.qemu.org/`.

[9] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis, "Path-exploration lifting: hi-fi tests for lo-fi emulators," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, (New York, NY, USA), pp. 337–348, ACM, 2012.

[10] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi, "Testing cpu emulators," in *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA'09)*, pp. 261–272, 2009.

[11] T. Raffetseder, C. Krügel, and E. Kirda, "Detecting system emulators," in *Information Security, 10th International Conference, ISC 2007*, pp. 1–18, October 2007.

[12] P. Ferrie, "Attacks on virtual machine emulators." Symantec Security Response, December 2006.

[13] "Kernel Based Virtual Machine." `http://www.linux-kvm.org/`.

[14] 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*, (Hyderabad, India), Dec. 2008.

[15] "TEMU: The BitBlaze dynamic analysis component." `http://bitblaze.cs.berkeley.edu/temu.html`.

[16] H. Yin and D. Song, "Temu: Binary code analysis via whole-system layered annotative execution," Tech. Rep. UCB/EECS-2010-3, EECS Department, University of California, Berkeley, Jan 2010.

[17] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, (Washington, DC, USA), pp. 297–312, IEEE Computer Society, 2011.

[18] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proceedings of Network and Distributed Systems Security Symposium (NDSS'03)*, February 2003.

[19] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction," in *Proceedings of the 14th ACM conference on Computer and Communications Security (CCS'07)*, October 2007.

[20] J. A. Goguen and J. Meseguer, "Security policies and security models," in *Proceedings of the 1982 IEEE Computer Society Symposium on Research in Security and Privacy*, (Oakland, CA), IEEE Computer Society Press, May 1982.

[21] J. Smith and R. Nair, "The architecture of virtual machines," *Computer*, vol. 38, no. 5, pp. 32–38, May.

[22] R. P. Goldberg, "Survey of virtual machine research," *Computer*, vol. 7, no. 6, pp. 34–45, June.

[23] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. Oracle America, Inc., 2011.

[24] D. Bornstein, "Dalvik vm internals." `https://sites.google.com/site/io/dalvik-vm-internals`, 2008. Google I/O.

[25] E. Mallach, "On the relationship between virtual machines and emulators," in *Proceedings of the Workshop on Virtual Computer Systems*, 1973.

[26] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," vol. 17, ACM, July 1974.

[27] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith, "Intel virtualization technology," *Computer*, vol. 38, pp. 48–56, May 2005.

[28] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-Xii, pp. 2–13, ACM, 2006.

[29] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig, "Intel virtualization technology: Hardware support for efficient processor virtualization," vol. 10, Intel, Aug. 2006.

[30] AMD, "AMD64 virtualization codenamed "pacifica" technology: Secure virtual machine architecture reference manual," Tech. Rep. 33047, Advanced Micro Devices, May 2005.

[31] R. Mijar and A. Nightingale, "Virtualization is coming to a platform near you," tech. rep., ARM Limited, 2011.

[32] Intel, *Intel 64 and IA-32 Architectures Software Developers Manual. Volume 3C: System Programming Guide, Part 3*, January 2013.

[33] N. Bhatia, "Performance evaluation of intel EPT hardware assist," tech. rep., VMware, Mar 2009.

[34] AMD, "AMD-V nested paging," tech. rep., Advanced Micro Devices, July 2008.

[35] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, pp. 164–177, October 2003.

[36] "Xen." `http://www.xen.org/`.

[37] "Oracle vm virtualbox." `http://www.virtualbox.org/`.

[38] "Android emulator." `http://developer.android.com/tools/help/emulator.html`.

[39] D. Brumley, C. Hartwig, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and D. Song, "BitScope: Automatically dissecting malicious binaries," Tech. Rep. CS-07-133, School of Computer Science, Carnegie Mellon University, Mar. 2007.

[40] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2011.

[41] 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(Oakland'07)*, May 2007.

[42] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *International Symposium on Code Generation and Optimization (CGO'03)*, March 2003.

[43] C.-K. 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 *Proc. of 2005 Programming Language Design and Implementation (PLDI) conference*, june 2005.

[44] K. Hazelwood and A. Klauser, "A dynamic binary instrumentation engine for the arm architecture," in *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, CASES '06, (New York, NY, USA), pp. 261–270, ACM, 2006.

[45] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *PLDI*, pp. 89–100, 2007.

[46] A. R. Bernat, K. Roundy, and B. P. Miller, "Efficient, sensitivity resistant binary instrumentation," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, (New York, NY, USA), pp. 89–99, ACM, 2011.

[47] A. Vasudevan and R. Yerraballi, "Cobra: Fine-grained malware analysis using stealth localized-executions," in *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*, (Washington, DC, USA), pp. 264–279, IEEE Computer Society, 2006.

[48] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Automatic reverse engineering of malware emulators," in *Proceedings of the 30th IEEE Symposium on Security and Privacy*, pp. 94–109, 2009.

[49] J. Caballero, N. M. Johnson, S. McCamant, and D. Song, "Binary code extraction and interface identification for security applications," in *Proceedings of the 17th Annual Network and Distributed System Security Symposium*, (San Diego, CA), Feb. 2010.

[50] U. Bayer, P. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering," in *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS 2009)*, 2009.

[51] J. Caballero, P. Poosankam, C. Kreibich, and D. Song, "Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering," in *Proceedings of the 16th ACM Conference on Computer and Communication Security*, (Chicago, IL), Nov. 2009.

[52] R. Riley, X. Jiang, and D. Xu, "Multi-aspect profiling of kernel rootkit behavior," in *Proceedings of the fourth ACM european conference on Computer systems (EuroSys'09)*, 2009.

[53] A. Lanzi, M. Sharif, and W. Lee, "K-Tracer: A system for extracting kernel malware behavior," in *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS'09)*, February 2009.

[54] H. Yin, Z. Liang, and D. Song, "HookFinder: Identifying and understanding malware hooking behaviors," in *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, February 2008.

[55] M. G. Kang, P. Poosankam, and H. Yin, "Renovo: A hidden code extractor for packed executables," in *Proceedings of the 5th ACM Workshop on Recurring Malcode (WORM'07)*, Oct. 2007.

[56] H. Yin, D. Song, E. Manuel, C. Kruegel, and E. Kirda, "Panorama: Capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS'07)*, October 2007.

[57] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song, "Dynamic Spyware Analysis," in *Proceedings of the 2007 Usenix Annual Conference (Usenix'07)*, June 2007.

[58] J. R. Crandall and F. T. Chong, "Minos: Control data attack prevention orthogonal to memory model," in *Proceedings of the 37th International Symposium on Microarchitecture (MICRO'04)*, December 2004.

[59] G. Portokalidis, A. Slowinska, and H. Bos, "Argos: an emulator for fingerprinting zero-day attacks," in *EuroSys 2006*, April 2006.

[60] 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*, pp. 51–62, 2008.

[61] P. P. Bungale and C.-K. Luk, "PinOS: a programmable framework for whole-system dynamic instrumentation," in *Proceedings of the 3rd international conference on Virtual execution environments*, VEE '07, pp. 137–147, 2007.

[62] G. Kroah-Hartman, "Driving me nuts - things you never should do in the kernel." `http://www.linuxjournal.com/article/8110`, April 2005.

[63] E. Mouw, "Faq/whywritingfilefromkernelisbad." `http://kernelnewbies.org/FAQ/WhyWritingFilesFromKernelIsBad`, November 2006.

[64] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, (Washington, DC, USA), pp. 317–331, IEEE Computer Society, 2010.

[65] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna, "Efficient Detection of Split Personalities in Malware," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, (San Diego, CA), February 2010.

[66] M. G. Kang, H. Yin, S. Hanna, S. McCamant, and D. Song, "Emulating emulation-resistant malware," in *Proceedings of the 2nd Workshop on Virtual Machine Security (VMSec'09)*, November 2009.

[67] "Gartner says sales of mobile devices grew 5.6 percent in third quarter of 2011; smartphone sales increased 42 percent." `http://gartner.com/it/page.jsp?id=1848514`, 2011.

[68] 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 *Proceedings of the 19th Network and Distributed System Security Symposium*, (San Diego, CA), February 2012.

[69] "Developing windows store apps." `http://msdn.microsoft.com/en-us/library/windows/apps/xaml/br229566.aspx`.

[70] E. Meijer and J. Gough, "A technical overview of the common language infrastructure." `http://research.microsoft.com/en-us/um/people/emeijer/Papers/CLR.pdf`, 2001.

[71] J. Singer, "JVM versus CLR: a comparative study," in *Proceedings of the 2nd international conference on Principles and practice of programming in Java*, PPPJ '03, (New York, NY, USA), 2003.

[72] "Security alert: New sophisticated android malware droidkungfu found in alternative chinese app markets." `http://www.csc.ncsu.edu/faculty/jiang/DroidKungFu.html`.

[73] "Cve-2009-1185." http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1185.

[74] "Android developers." http://developer.android.com/.

[75] "ded: Decompiling Android Applications." http://siis.cse.psu.edu/ded/index.html.

[76] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security," in *Proceedings of the 20th USENIX Security Symposium*, 2011.

[77] "Proguard." http://proguard.sourceforge.net.

[78] "Dynamic, metamorphic (and opensource) virtual machines." http://archive.hack.lu/2010/Desnos_Dynamic_Metamorphic_Virtual_Machines-slides.pdf.

[79] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, (Berkeley, CA, USA), pp. 1–6, USENIX Association, 2010.

[80] "Droidbox: Android application sandbox." http://code.google.com/p/droidbox/.

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

[82] J. Oberheide and C. Miller, "Dissecting the android bouncer." http://jon.oberheide.org/2012/06/21/dissecting-the-android-bouncer/, June 2012.

[83] J. Caballero, H. Yin, Z. Liang, and D. Song, "Polyglot: Automatic extraction of protocol message format using dynamic binary analysis," in *Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS'07)*, October 2007.

[84] J. Seward and N. Nethercote, "Using valgrind to detect undefined value errors with bit-precision," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, (Berkeley, CA, USA), pp. 2–2, USENIX Association, 2005.

[85] J. Clause and A. Orso, "Leakpoint: pinpointing the causes of memory leaks," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, (New York, NY, USA), pp. 515–524, ACM, 2010.

[86] S. McCamant and M. D. Ernst, "Quantitative information flow as network flow capacity," in *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, (New York, NY, USA), pp. 193–205, ACM, 2008.

[87] L. C. Lam and T. cker Chiueh, "A general dynamic information flow tracking framework for security applications," in *Computer Security Applications Conference, 2006. ACSAC '06. 22nd Annual*, pp. 463–472, dec. 2006.

[88] W. Chang, B. Streiff, and C. Lin, "Efficient and extensible security enforcement using dynamic data flow analysis," in *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, (New York, NY, USA), pp. 39–50, ACM, 2008.

[89] I. Papagiannis, M. Migliavacca, and P. Pietzuch, "PHP aspis: using partial taint tracking to protect against injection attacks," in *Proceedings of the 2nd USENIX conference on Web application development*, WebApps'11, (Berkeley, CA, USA), pp. 2–2, USENIX Association, 2011.

[90] W. Xu, S. Bhatkar, and R. Sekar, "Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks," in *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, (Berkeley, CA, USA), USENIX Association, 2006.

[91] W. G. J. Halfond, A. Orso, and P. Manolios, "Using positive tainting and syntax-aware evaluation to counter sql injection attacks," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, (New York, NY, USA), pp. 175–185, ACM, 2006.

[92] E. Chin and D. Wagner, "Efficient character-level taint tracking for java," in *Proceedings of the 2009 ACM workshop on Secure web services*, SWS '09, (New York, NY, USA), pp. 3–12, ACM, 2009.

[93] B. Chess and J. West, "Dynamic taint propagation: Finding vulnerabilities without attacking," *Inf. Secur. Tech. Rep.*, vol. 13, pp. 33–39, Jan. 2008.

[94] J. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework," in *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, (New York, NY, USA), pp. 196–206, ACM, 2007.

[95] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "libdft: practical dynamic data flow tracking for commodity systems," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, VEE '12, (New York, NY, USA), pp. 121–132, ACM, 2012.

[96] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall, "Tainteraser: protecting sensitive data leaks using application-level taint tracking," *SIGOPS Oper. Syst. Rev.*, vol. 45, pp. 142–154, Feb. 2011.

[97] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu, "Lift: A low-overhead practical information flow tracking system for detecting security attacks," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, (Washington, DC, USA), pp. 135–148, IEEE Computer Society, 2006.

[98] A. Nguyen-tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, "Automatically hardening web applications using precise tainting," in *In 20th IFIP International Information Security Conference*, pp. 372–382, 2005.

[99] V. Haldar, D. Chandra, and M. Franz, "Dynamic taint propagation for java," in *Proceedings of the 21st Annual Computer Security Applications Conference*, ACSAC '05, (Washington, DC, USA), pp. 303–311, IEEE Computer Society, 2005.

[100] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis," in *Proceeding of the Network and Distributed System Security Symposium (NDSS'07)*, February 2007.

[101] "perlsec: Taint mode." `perldoc.perl.org/perlsec.html#Taint-mode`, August 2012.

[102] 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)*, February 2005.

[103] E. Bosman, A. Slowinska, and H. Bos, "Minemu: The world's fastest taint tracker," in *Proceedings of RAID'11*, (Menlo Park, CA), September 2011.

[104] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "DTA++: Dynamic taint analysis with targeted control-flow propagation," in *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, (San Diego, CA), Feb. 2011.

[105] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August, "Rifle: An architectural framework for user-centric information-flow security," in *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, (Washington, DC, USA), pp. 243–254, IEEE Computer Society, 2004.

[106] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)*, October 2004.

[107] M. Ozsoy, D. Ponomarev, N. Abu-Ghazaleh, and T. Suri, "Sift: a low-overhead dynamic information flow tracking architecture for smt processors," in *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, (New York, NY, USA), pp. 37:1–37:11, ACM, 2011.

[108] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," in *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, (New York, NY, USA), pp. 109–120, ACM, 2009.

[109] J. Kong, C. C. Zou, and H. Zhou, "Improving software security via runtime instruction-level taint checking," in *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, ASID '06, (New York, NY, USA), pp. 18–24, ACM, 2006.

[110] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. Iyer, "Defeating memory corruption attacks via pointer taintedness detection," in *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pp. 378 – 387, june-1 july 2005.

[111] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: a flexible information flow architecture for software security," in *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, (New York, NY, USA), pp. 482–493, ACM, 2007.

[112] D. M. Volpano, "Safety versus secrecy," in *Proceedings of the 6th International Symposium on Static Analysis*, SAS '99, (London, UK, UK), pp. 303–311, Springer-Verlag, 1999.

[113] T. Bao, Y. Zheng, Z. Lin, X. Zhang, and D. Xu, "Strict control dependence and its effect on dynamic information flow analyses," in *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, (New York, NY, USA), pp. 13–24, ACM, 2010.

[114] A. Slowinska and H. Bos, "Pointless tainting?: evaluating the practicality of pointer tainting," in *EuroSys '09*, April 2009.

[115] M. Olszewski, J. Ansel, and S. Amarasinghe, "Kendo: efficient deterministic multithreading in software," *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*, Mar. 2009.

[116] D. Geels, G. Altekar, S. Shenker, and I. Stoica, "Replay debugging for distributed applications," in *Proceedings of the 2006 USENIX Annual Technical Conference*, pp. 27–27, 2006.

[117] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang, "R2: An application-level kernel for record and replay," in *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI'08)*, pp. 193–208, 2008.

[118] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou, "Flashback: a lightweight extension for rollback and deterministic replay for software debugging," in *Proceedings of the 2004 USENIX Annual Technical Conference*, June 2004.

[119] Y. Saito, "Jockey: a user-space library for record-replay debugging," in *AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pp. 69–76, 2005.

[120] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "Revirt: Enabling intrusion analysis through virtual-machine logging and replay," in *Proceedings of the 5th symposium on Operating Systems Design and Implementation (OSDI'02)*, December 2002.

[121] "Vmware." `http://www.vmware.com/`.

[122] J. Chow, T. Garfinkel, and P. Chen, "Decoupling dynamic program analysis from execution in virtual environments," in *Proceedings of 2008 Usenix Annual Technical Conference*, June 2008.

[123] C. da Wang and S. Ju, "The dilemma of covert channels searching," in *ICISC*, pp. 169–174, 2005.

[124] M. Siper, *Introduction to the Theory of Computation*. International Thomson Publishing, 1996.

[125] "adore-ng rootkit." `http://stealth.openwall.net/rootkits/`.

[126] "Anubis: Analyzing Unknown Binaries." `http://anubis.iseclab.org/`.

[127] "CWSandbox::Behavior-based Malware Analysis." `http://mwanalysis.org/`.

[128] B. Cheng and B. Buzbee, "A JIT compiler for android's dalvik VM." `http://www.google.com/events/io/2010/sessions/jit-compiler-androids-dalvik-vm.html`, 2010. Google I/O.

[129] L. De Moura and N. Bjørner, "Z3: an efficient smt solver," in *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, (Berlin, Heidelberg), pp. 337–340, Springer-Verlag, 2008.

[130] "Jif: Java + information flow." `http://www.cs.cornell.edu/jif/`, August 2012.

[131] A. C. Myers, "Jflow: practical mostly-static information flow control," in *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, (New York, NY, USA), pp. 228–241, ACM, 1999.

[132] F. Pottier and V. Simonet, "Information flow inference for ml," *ACM Trans. Program. Lang. Syst.*, vol. 25, pp. 117–158, Jan. 2003.

[133] "Valgrind: Project suggestions." `http://valgrind.org/help/projects.html`, July 2012.

[134] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB Standard: Version 2.0," in *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)* (A. Gupta and D. Kroening, eds.), 2010.

[135] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A binary analysis platform," in *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, (Berlin, Heidelberg), pp. 463–469, Springer-Verlag, 2011.

[136] P. Godefroid and A. Taly, "Automated synthesis of symbolic instruction encodings from i/o samples," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, (New York, NY, USA), pp. 441–452, ACM, 2012.

[137] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, pp. 319–349, July 1987.

[138] K. Jee, G. Portokalidis, V. P. Kemerlis, and S. Ghosh, "A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware," in *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, (San Diego, CA), Feb. 2012.

[139] J. Newsome, S. McCamant, and D. Song, "Measuring channel capacity to distinguish undue influence," in *Proceedings of the Fourth ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, (Dublin, Ireland), June 2009.

[140] D. Devriese and F. Piessens, "Noninterference through secure multi-execution," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, (Washington, DC, USA), pp. 109–124, IEEE Computer Society, 2010.

[141] W. D. Groef, D. Devriese, N. Nikiforakis, and F. Piessens, "Flowfox: a web browser with flexible and precise information flow control," in *Proceedings of the 19th ACM Conferences on Computer and Communication Security (CCS'12)*, October 2012.

VITA

VITA

Lok Kwong Yan was born in a village named Shiukou Cun in Huadu, Guangdong, China. He received his Bachelor of Science degree in Computer Engineering and Master of Science degree in Electrical engineering from Polytechnic University, now known as Polytechnic Institute of New York University. He received his PhD in Computer Information Science and Engineering from Syracuse University in May 2013.