Towards Transparent Debugging

Fengwei Zhang, Kevin Leach, Angelos Stavrou, and Haining Wang

Abstract—Traditional malware analysis relies on virtualization or emulation technology to run samples in a confined environment, and to analyze malicious activities by instrumenting code execution. However, virtual machines and emulators inevitably create artifacts in the execution environment, making these approaches vulnerable to detection or subversion. In this paper, we present MalT, a debugging framework that employs System Management Mode, a CPU mode in the x86 architecture, to transparently study armored malware. MalT does not depend on virtualization or emulation and thus is immune to threats targeting such environments. Our approach reduces the attack surface at the software level, and advances state-of-the-art debugging transparency. MalT embodies various debugging functions, including register/memory accesses, breakpoints, and seven stepping modes. Additionally, MalT restores the system to a clean state after a debugging session. We implemented a prototype of MalT on two physical machines, and we conducted experiments by testing an array of existing anti-virtualization, anti-emulation, and packing techniques against MalT. The experimental results show that our prototype remains transparent and undetected against the samples. Furthermore, debugging and restoration introduce moderate but manageable overheads on both Windows and Linux platforms.

Index Terms—Malware debugging, transparency, SMM

1 Introduction

Traditional malware analysis employs virtualization [1], [2], [3] and emulation [4], [5], [6] technologies to dissect malware behavior at runtime. This approach runs the malware in a Virtual Machine (VM) or emulator and uses an analysis program to introspect the malware from the outside so that the malware cannot infect the analysis program. Unfortunately, malware writers can easily escape this analysis mechanism by using a variety of anti-debugging, anti-virtualization, and anti-emulation techniques [7], [8], [9], [10], [11], [12]. Malware can easily detect the presence of a VM or emulator and alter its behavior to hide itself. Chen et al. [7] executed 6,900 malware samples and found that more than 40 percent of them reduced malicious behavior under a VM or with a debugger attached. Branco et al. [8] showed that 88 and 81 percent of 4 million analyzed malware samples had anti-reverse engineering and anti-virtualization techniques, respectively. Furthermore, Garfinkel et al. [13] concluded that virtualization transparency is fundamentally infeasible and impractical. To address this problem, security researchers have proposed analyzing malware on bare metal [14], [15]. This approach makes anti-VM malware expose its malicious behavior, and it does not require any virtualization or emulation technology. However, previous bare-metal malware analysis [14] depends on the OS, and ring 0 malware can evade the analysis. Thus, stealthy malware detection and analysis still remains an open research problem.

In this paper, we present MalT, a novel approach that progresses towards stealthy debugging by leveraging System Management Mode (SMM) to transparently debug software on bare metal. Our system is motivated by the intuition that malware debugging needs to be transparent, and it should not leave artifacts introduced by the debugging functions. SMM is a special-purpose CPU mode in all x86 platforms. The main benefit of SMM is to provide a distinct and easily isolated processor environment that is transparent to the OS or running applications. With the help of SMM, we are able to achieve a high level of transparency, which enables a strong threat model for malware debugging. We briefly describe its basic workflow as follows. We run malware on one physical target machine and employ SMM to communicate with the debugging client on another physical machine. While SMM executes, Protected Mode is essentially paused. The OS and hypervisor, therefore, are unaware of code executing in SMM. Because we run debugging code in SMM, we expose far fewer artifacts to the malware, enabling a more transparent execution environment for the debugging code than existing approaches.

The debugging client communicates with the target server using a GDB-like protocol with serial messages. We implement the basic debugging commands (e.g., breakpoints and memory/register examination) in the current prototype of MalT. Furthermore, we implement seven techniques to provide step-by-step debugging that includes instruction-level, branch-level, far control transfer level, and near return transfer level. Additionally, because running a malware sample changes the system's state and these changes can lead to an incorrect runtime result of the next malware sample, we implement a novel technique to remotely restore the target system to a clean state after a debugging session. It uses SMM to reliably restore the hard disk so that even ring 0 malware cannot tamper with the restoration process.

- *F. Zhang is with the Department of Computer Science, Wayne State University, Detroit, MI 48202. E-mail: fengwei@wayne.edu.*
- *K. Leach is with the Department of Computer Science, University of Virginia, Charlottesville, VA 22904. E-mail: kjl2y@virginia.edu.*
- *A. Stavrou is with the Department of Computer Science, George Mason University, Fairfax, VA 22030. E-mail: astavrou@gmu.edu.*
- *H. Wang is with the Department of Electrical and Computer Engineering, University of Delaware, Newark, DE 19716. E-mail: hnw@udel.edu.*

*Manuscript received 14 July 2015; revised 3 Mar. 2016; accepted 15 Mar. 2016. Date of publication 23 Mar. 2016; date of current version 9 Mar. 2018.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TDSC.2016.2545671*

MalT runs the debugging code in SMM without using a hypervisor. Thus, it has a smaller Trusted Code Base (TCB) than hypervisor-based debugging systems [1], [4], [5], [6]. Moreover, MalT is OS-agnostic and immune to hypervisor attacks (e.g., VM-escape attacks [16], [17]). Compared to existing bare-metal malware analysis [14], [15], SMM has the same privilege level as hardware. Thus, MalT is capable of debugging and analyzing kernel and hypervisor rookits as well [18], [19].

We develop a prototype of MalT on two physical machines connected by a serial cable. To demonstrate the efficiency and transparency of our approach, we test MalT with popular packing, anti-debugging, anti-virtualization, and anti-emulation techniques. The experimental results show that MalT remains transparent against these techniques. Additionally, our experiments demonstrate that MalT is able to debug crashed kernels/hypervisors. MalT introduces a reasonable overhead: It takes about 12 microseconds on average to execute the debugging code without command communication. Moreover, we use popular benchmarks to measure the performance overhead for the seven types of step-by-step execution on Windows and Linux platforms. The overhead ranges from 1.46 to 1519 times slowdown on the target system, depending on the user's selected instrumentation method. In addition, we measure the time for restoring the target system, and it takes about 102 seconds to completely restore the system to a clean state. The main contributions of this work are:

- We provide a bare-metal debugging tool called MalT that leverages SMM for malware analysis. It leaves a minimal footprint on the target machine and provides a more transparent execution environment for the debugger than existing approaches.
- We introduce a hardware-assisted malware analysis approach that does not use the hypervisor and OS code. MalT is OS-agnostic and is capable of conducting hypervisor rootkit analysis and kernel debugging.
- We implement various debugging functions, including breakpoints and step-by-step debugging. Our experiments demonstrate that MalT induces moderate but manageable overhead on Windows and Linux environments.
- We implement a novel technique to completely and reliably restore the target system to a clean state after a debugging session.
- Through testing MalT against popular packers, anti-debugging, anti-virtualization, and anti-emulation techniques, we demonstrate that MalT remains transparent and undetected.

This paper is an extended version of our previous work [20] published in IEEE Security and Privacy 2015 and is organized as follows. Section 2 provides background on SMM and BIOS. Section 3 surveys related work. Section 4 discusses our threat model and assumptions. Section 5 presents the architecture of MalT. Section 6 details the implementation of MalT. Section 7 analyzes the transparency of MalT. Section 8 shows the performance evaluation of our prototype. Section 9 discusses the limitations of MalT. Section 10 concludes the paper.

2 Background

2.1 System Management Mode

System Management Mode [21] is a mode of execution similar to Real and Protected modes available on x86 platforms. It provides a transparent mechanism for implementing platform-specific system control functions such as power management. It is setup by the Basic Input/Output System (BIOS) that is responsible for initializing the hardware during the booting process.

SMM is triggered by asserting the System Management Interrupt (SMI) pin on the CPU. This pin can be asserted in a variety of ways, which include writing to a hardware port or generating Message Signaled Interrupts with a PCI device. Next, the CPU saves its state to a special region of memory called System Management RAM (SMRAM). Then, it atomically executes the SMI handler stored in SMRAM. SMRAM cannot be addressed by the other modes of execution. The requests for addresses in SMRAM are instead forwarded to video memory by default. This caveat therefore allows SMRAM to be used as secure storage. The SMI handler is loaded into SMRAM by the BIOS at boot time. The SMI handler has unrestricted access to the physical address space and can run any instructions requiring any privilege level. SMM is often referred to as *ring -2* since the OS is referred as *ring 0* and hypervisors are referred to as *ring -1*. The `RSM` instruction forces the CPU to exit from SMM and resume execution in the previous mode.

2.2 BIOS and Coreboot

The BIOS is an integral part of a computer. It initializes hardware and loads the operating system. The BIOS code is stored on non-volatile memory on the motherboard. In particular, we make use of an open-source BIOS called Coreboot [22]. Coreboot performs some hardware initialization and then executes a payload (e.g., UEFI). MalT uses SeaBIOS [23] as the Coreboot payload. Coreboot is written mostly in C and allows us to edit the SMI handler very easily. This makes MalT much more portable as Coreboot abstracts away the heterogeneity of specific hardware configurations.

3 Related Works

3.1 Malware Debugging and Analysis

VAMPiRE [24] is a software breakpoint framework running within the operating system. Since it has the same privilege level as the operating system kernel, it can only debug ring three malware. Rootkits can gain kernel-level privileges to circumvent VAMPiRE. However, as MalT does not rely on the operating system, it can debug rootkits safely.

Ether [1] is a malware analysis framework based on hardware virtualization extensions (e.g., Intel VT). It runs outside of the guest operating systems by relying on underlying hardware features. BitBlaze [26] and Anubis [5] are QEMU-based malware analysis systems. They focus on understanding malware behaviors, instead of achieving better transparency. V2E [4] combines both hardware virtualization and software emulation. HyperDbg [3] uses the hardware virtualization that allows the late launching of VMX modes to install a virtual machine monitor and run the analysis code in the VMX root mode. SPIDER [2] uses Extended Page Tables to implement invisible breakpoints

TABLE 1
Comparison with Other Debuggers

	MALT	**BareBox** [14]	**V2E** [4]	**Anubis** [5]	**Virt-ICE** [6]	**Ether** [1]	**HyperDbg** [3]	**VAMPiRE** [24]	**SPIDER** [2]	**IDAPro** [25]
No VM/emulator	✓	✓						✓		✓
Debug ring0 malware	✓		✓	✓	✓	✓	✓		✓	
Trusted code base	***BIOS***	OS	KVM+QEMU	QEMU	QEMU	Xen	HF	OS	KVM	OS
SLOC of TCB (K)	**1.5**	16,281	13,397	786	786	509	18	16,281	12,593	16,281

and hardware virtualization to hide its side effects. Compared to our system, Ether, BitBlaze, Anubis, V2E, HyperDbg, and SPIDER all rely on easily detected emulation or virtualization technology [7], [10], [12], [27] and make the assumption that virtualization or emulation is transparent from guest-OSes. In contrast, MALT relies on the BIOS code to analyze malware on the bare metal. Additionally, nEther [28] has demonstrated that malware running in the guest OS can detect the presence of Ether using CPUID bits, while MALT remains transparent. Moreover, MALT has a smaller trusted computing base than hypervisor-based malware analysis systems. Table 1 shows the trusted computing base of various malware analysis systems.

Virt-ICE [6] is a remote debugging framework similar to MALT. It leverages emulation technology to debug malware in a VM and communicates with a debugging client over a TCP connection. As it debugs the system outside of the VM, it is capable of analyzing rootkits and other ring 0 malware transparently. Willems et al. [15] used branch tracing to record all the branches taken by a program execution. As pointed out in the paper, the data obtainable by branch tracing is rather coarse, and this approach still suffers from a CPU register attack against branch tracing settings. However, MALT provides fine-grained debugging methods and can defend against mutation of CPU registers. BareCloud [29] is a recent armored malware detection system; it executes malware on a bare-metal system and compares disk- and network-activities of the malware with other emulation and virtualization-based analysis systems for evasive malware detection, while MALT is used for malware debugging. Table 1 summarizes the differences between MALT and other malware debugging and analysis systems. The source lines of code (SLOC) of HyperDbg Framework (HF) is calculated by running `wc` command on its source code. Other SLOCs are obtained from [30], and we use the Linux kernel as the OS in Table 1.

3.2 System Restoration

BareBox [14] is a malware analysis framework based on a bare-metal machine without any virtualization or emulation technologies. It uses a small OS running outside of the target OS and provides a fast approach to restore a system without rebooting. However, MALT improves BareBox in the following two aspects: *Completeness* and *High-privilege*. Completeness refers to all components in the target system that must be restored. They include registers, cache, memory, disks, I/O devices, and firmware like BIOS. BareBox only reinitializes some devices by manipulating the power state, while MALT uses Coreboot [22] to reinitialize all devices. Additionally, MALT restores firmware by flashing the BIOS and other devices. High-privilege means malware with ring 0 privilege cannot tamper with or modify the restoring process. BareBox only works for user-level malware, and malware with higher privilege can easily affect the Meta-OS and tamper with the restoration process such as re-imaging disk. However, MALT uses SMM to reliably restore the disk so that ring 0 rootkits cannot tamper with the restoration. Other system restoration approaches [31], [32] use virtualization technology, which violates the transparency requirement in our threat model.

3.3 SMM-Based Systems

In recent years, SMM-based research has appeared in the security literature. For instance, SMM can be used to check the integrity of higher level software (e.g., hypervisor and OS). HyperGuard [33], HyperCheck [34], and HyperSentry [35] are integrity monitoring systems based on SMM. SPECTRE [36] uses SMM to introspect the live memory of a system for malware detection. Another use of SMM is to reliably acquire system physical memory for forensic analysis [37], [38]. However, MALT differs from previous SMM-based systems in these aspects: (1) MALT is the first system that uses SMM for debugging, and its intended usage involves with human interaction; (2) it addresses the debugging transparency problem by mitigating its side effects, while previous systems do not consider this challenging problem; (3) it uses a variety of methods to trigger SMIs, and the triggering frequency can be instruction-level. In addition, other security researchers have proposed using SMM to implement attacks. In 2004, Duflot [39] demonstrated the first SMM-based attack to bypass the protection mechanism in OpenBSD. Other SMM-based attacks focus on achieving stealthy rootkits [40], [41]. For instance, the National Security Agency (NSA) uses SMM to build an array of rootkits including DEITYBOUNCE for Dell and IRONCHEF for HP Proliant servers [42].

4 THREAT MODEL AND ASSUMPTIONS

4.1 Usage Scenarios

MALT is intended to transparently analyze a variety of code that is capable of detecting or disabling typical malware analysis or detection tools. We consider two types of powerful malware in our threat model: armored malware and rootkits.

4.1.1 Armored Malware

Armored malware or evasive malware [29] is a piece of code that employs anti-debugging techniques. Malicious code can be made to alter its behavior if it detects the presence of a debugger. There are many different detection techniques employed by current malware [9]. For example, `IsDebuggerPresent()` and `CheckRemoteDebuggerPresent()` are Windows API methods in the kernel32 library returning values based upon the presence of a debugger. Legitimate software developers can take advantage of such API calls to ease the debugging process in their own software. However, malware can use these methods to determine if it is being debugged to change or hide its malicious behavior from analysis.

Malware can also determine if it is running in a virtual machine or emulator [7], [11], [12]. For instance, Red Pill [27] can efficiently detect the presence of a VM. It executes a non-privileged (*ring 3*) instruction, `SIDT`, which reads the value stored in the Interrupt Descriptor Table (IDT) register. The base address of the IDT will be different in a VM than on a bare-metal machine because there is only one IDT register shared by both host-OS and guest-OS. Additionally, QEMU can be detected by accessing a reserved Model Specific Register (MSR) [4]. This invalid access causes a General Protection (GP) exception on a bare-metal machine, but QEMU does not. Note that some kinds of detection are due to fundamental limitations (e.g., `SIDT` is not a privileged instruction) while others are implementation gaps (e.g., MSRs in QEMU).

4.1.2 Rootkits

Rootkits are a type of stealthy malicious software. Specifically, they hide certain process information to avoid detection while maintaining continued privileged access to a system. There are a few types of rootkits ranging from user mode to firmware level. For example, kernel mode rootkits run in the OS kernel (ring 0) by modifying the kernel code or kernel data structures (e.g., Direct Kernel Object Modification). Hypervisor-level rootkits run in *ring -1* and host the target operating system as a virtual machine. These rootkits intercept all of the operations including hardware calls in the target OS, as shown in Subvirt [18] and BluePill [19]. Since MALT runs in SMM with *ring -2* privilege, it is capable of debugging user mode, kernel mode, and hypervisor-level rootkits. As no virtualization is used, MALT is immune to hypervisor attacks (e.g., VM escape [16], [17]). However, firmware rootkits running in *ring -2* are out of the scope.

4.2 Assumptions

As our trusted code (SMI handler) is stored in the BIOS, we assume the BIOS will not be compromised. We assume the Core Root of Trust for Measurement (CRTM) is trusted so that we can use Static Root of Trust for Measurement (SRTM) to perform the self-measurement of the BIOS and secure the boot process [43]. We also assume the firmware is trusted, although we can use SMM to check its integrity [44]. After booting, we lock the SMRAM to ensure the SMI handler code is trusted. We discuss attacks against SMM in Section 9. We assume the debugging client and remote machine are trusted. We assume the system starts without malicious code but an attacker can exploit software vulnerabilities to gain the control of the system. Furthermore, we consider an attacker that can have unlimited computational resources on our machine. We assume the attacker compromises the OS with a maximal speed Last, we assume the attacker does not have physical access to the machines. Malicious hardware (e.g., hardware trojans) is also out of scope.

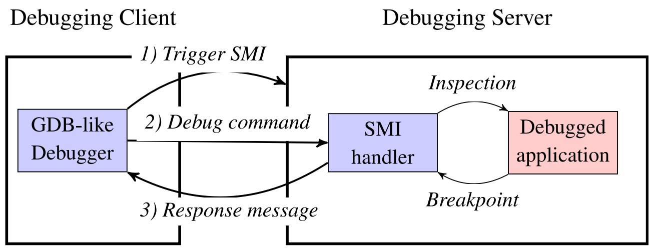


Fig. 1. Architecture of MALT.

5 System Architecture

Fig. 1 shows the architecture of the MALT system. The debugging client is equipped with a simple GDB-like debugger (debugging commands have similar syntax to GDB's). The user inputs basic debugging commands (e.g., *list registers*), and then the target machine executes the command and replies to the client as required. When a command is entered, the client sends a message via a serial cable to the debugging server. This message contains the actual command. While in SMM, the debugging server transmits a response message containing the information requested by the command. Since the target machine executes the actual debugging command within the SMI handler, its operation remains transparent to the target application and underlying operating system.

As shown in Fig. 1, the debugging client first sends an SMI triggering message to the debugging server; MALT reroutes a serial interrupt to generate an SMI when the message is received. Second, once the debugging server enters SMM, the debugging client starts to send debugging commands to the SMI handler on the server. Third, the SMI handler transparently executes the requested commands (e.g., list registers and set breakpoints) and sends a response message back to the client.

The SMI handler on the debugging server inspects the debugged application at runtime. If the debugged application hits a breakpoint, the SMI handler sends a breakpoint hit message to the debugging client and stays in SMM until further debugging commands are received. Once SMM has control of the system, we configure the next SMI via performance counters on the CPU. Next, we detail each component of the MALT system.

5.1 Debugging Client

The client can ideally implement a variety of popular debugging options. For example, we could use the SMI handler to implement the GDB protocol so that it would properly interface with a regular GDB client. Similarly, we might implement the necessary plugin for IDAPro to correctly interact with our system. Our related conference publication [20] discusses the prospect of combining MALT with GDB and IDAPro. However, this would require implementing a complex protocol within the SMI handler, which we leave for future work. Instead, we implement a custom protocol with which to communicate between the debugging client and the SMI handler. MALT implements a small GDB-like client to simplify our implementation. For the system restoration process, we store a clean disk image on the debugging client machine. Section 6.7 explains this in detail.

5.2 Debugging Server

The debugging server consists of two parts—the SMI handler and the debugging target that contains the malware-infected OS and applications. The SMI handler implements the critical debugging features (e.g., breakpoints and state reports), thus restricting the execution of debugging code to System Management Mode. The debugging target executes in Protected Mode or its other usual execution mode. Since the CPU state is saved within SMRAM when switching to SMM, we can reconstruct useful information and perform typical debugging operations each time an SMI is triggered.

SMRAM contains architectural state information of the thread that was running when the SMI was triggered. Since the SMIs are produced regardless of the running thread, SMRAM often contains a state unrelated to the debugging target. In order to find the relevant state information, we must solve the well-known semantic gap problem. By bridging the semantic gap within the SMI handler, we can ascertain the state of the thread executing in Protected Mode. This is similar to Virtual Machine Introspection (VMI) systems [45]. We need to continue our analysis in the SMI handler only if the SMRAM state belongs to a thread we are interested in debugging. Otherwise, we can exit the SMI handler immediately. Note that MALT does not require Protected Mode; SMM can be initialized from other x86 modes (e.g., Real Mode), but the semantics of the code would be different.

5.3 Communication

In order to implement remote debugging in our system, we define a simple communication protocol used by the client and server hosts. The detailed protocol is described in our previous paper [20]. The commands are derived from basic GDB stubs, which are intended for debugging embedded software. The commands cover the basic debugging operations upon which the client can expand. The small number of commands greatly simplifies the process of communication within the SMI handler. For the disk restoration process, the debugging server sends a chunk location to a remote machine, and the remote machine responds with the chunk value. Section 6.7 details this further.

6 DESIGN AND IMPLEMENTATION

6.1 Debugging Client

The client machine consists of a simple command line application. A user can direct the debugger to perform useful tasks, such as setting breakpoints. For example, the user writes simple commands such as `b 0xdeadbeef` to set a breakpoint at address `0xdeadbeef`. The specific commands are described in our previous paper [20]. We did not implement features such as symbols. The client machine uses serial messages to communicate with the server.

6.2 Debugging Server

The target machine consists of a computer with a custom Coreboot-based BIOS. We changed the SMI handler in the Coreboot code to implement a simple debugging server. This custom SMI handler is responsible for all typical debugging functions found in other debuggers such as GDB. We implemented remote debugging functions via the serial protocol to achieve common debugging functions such as breakpoints, step-by-step execution, and state inspection and mutation.

6.3 Semantic Gap Reconstruction

As with VMI systems [46], SMM-based systems encounter the well-known semantic gap problem. In brief, code running in SMM cannot understand the semantics of raw memory. The CPU state saved by SMM only belongs to the thread that was running when the SMI was triggered. If we use step-by-step execution, there is a chance that another application is executing when the SMI occurs. Thus, we must be able to identify the target application so that we do not interfere with the execution of unrelated applications. This requires reconstructing OS semantics. Note that MALT has the same assumptions as traditional VMI systems [45].

In Windows, we start with the Kernel Processor Control Region (`KPCR`) structure associated with the CPU, which has a static linear address, 0xffdff000. At offset 0×34 of `KPCR`, there is a pointer to another structure called `KdVersionBlock`, which contains a pointer to `PsActiveProcessHead`. The `PsActiveProcessHead` serves as the head of a doubly and circularly linked list of Executive Process (`EProcess`) structures. The `EProcess` structure is a process descriptor containing critical information for bridging the semantic gap in Windows NT kernels.

In particular, the Executive Process contains the value of the CR3 register associated with the process. The value of the CR3 register contains the physical address of the base of the page table of that process. We use the name field in the `EProcess` to identify the CR3 value of the target application when it executes first instruction. Since malware may change the name field, we only compare the saved CR3 with the current CR3 to identify the target process for further debugging. Filling the semantic gap in Linux is a similar procedure, but there are fewer structures and thus fewer steps. Previous works [36], [47] describe the method, which MALT uses to debug applications on the Linux platform. Note that malware with ring 0 privilege can manipulate the kernel data structures to confuse the reconstruction process, and current semantic gap solutions suffer from this limitation [45]. As with VMI systems, MALT assumes that malware does not mutate kernel structures for correctly bridging the semantic gap.

6.4 Triggering an SMI

The system depends upon reliable assertions of System Management Interrupts (SMIs). Because the debugging code is placed in the SMI handler, it will not work unless the CPU can stealthily enter SMM.

In general, we can assert an SMI via software or hardware. The software method writes to an Advanced Configuration and Power Interface (ACPI) port to trigger an SMI, and we can use this method to implement software breakpoints. We can place an `out` instruction in the malware code so that when the malware's control flow reaches that point, SMM begins execution, and the malware can be analyzed. The assembly instructions are: `mov $0x52f, %dx; out %ax, (%dx);` The first instruction moves the SMI software interrupt port number (0×2b on Intel, and 0×52f in

TABLE 2
Stepping Methods in MalT

Performance Counter Events	**Description** [49]
Retired instructions	Counts retired instructions, plus exceptions and interrupts (each count as one instruction)
Retired branch	Includes all types of architectural control flow changes, including exceptions and interrupts
Retired mispredicted branch	Counts the number of branch retired that were not correctly predicted
Retired taken branches	Counts the number of taken branches that were retired
Retired taken branch mispredicted	Counts number of retired taken branch instructions that were mispredicted
Retired far control transfers	Includes far calls/jumps/returns, IRET, SYSCALL and SYSRET, exceptions and interrupts
Retired near returns	Counts near return instructions (RET or RET Iw) retired

our chipset [48]) into the `dx` register, and the second instruction writes the contents stored in `ax` to that SMI software interrupt port. (The value stored in `ax` is inconsequential). In total, these two instructions take six bytes: `66 BA 2F 05 66 EE`. While this method is straightforward, it is similar to traditional debuggers using INT3 instructions to insert arbitrary breakpoints. The alternative methods described below are harder to detect by self-checking malware.

In MalT, we use two hardware-based methods to trigger SMIs. The first uses a serial port to trigger an SMI to start a debugging session. In order for the debugging client to interact with the debugging server and start a session, we reroute a serial interrupt to generate an SMI by configuring the redirection table in I/O Advanced Programmable Interrupt Controller (APIC). We use serial port COM1 on the debugging server, and its Interrupt Request (IRQ) number is 4. We configure the redirection table entry of IRQ 4 at offset `0x18` in I/O APIC and change the Delivery Mode (DM) to be SMI. Therefore, an SMI is generated when a serial message arrives. The debugging client sends a triggering message, causing the target machine to enter SMM. Once in SMM, the debugging client sends further debugging commands to which the target responds. In MalT, we use this method to trigger the first SMI and start a debugging session on the debugging server. The time of triggering the first SMI is right before each debugging session after reboot, because MalT assumes that the first instruction of malware can compromise the system.

The second hardware-based method uses performance counters to trigger an SMI. This method leverages two architectural components of the CPU: performance monitoring counters and Local Advanced Programmable Interrupt Controller (LAPIC) [49]. First, we configure the Performance Counter Event Selection (PerfEvtSel0) register to select the counting event. There is an array of events from which to select; we use different events to implement various debugging functionalities. For example, we use the Retired Instructions Event (C0h) to single-step the whole system. Next, we set the corresponding performance counter (PerfCtr0) register to the maximum value. In this case, if the selected event happens, it overflows the performance counter. Last, we configure the Local Vector Table Entry (LVTE) in LAPIC to deliver SMIs when an overflow occurs. Similar methods [35], [50] are used to switch from a guest VM to the hypervisor VMX root mode.

6.5 Breakpoints

Breakpoints are generally software- or hardware-based. Software breakpoints allow for unlimited breakpoints, but they must modify a program's code, typically placing a single interrupt or trap instruction at the breakpoint. Self-checking malware can easily detect or interfere with such changes. On the other hand, hardware breakpoints do not modify code, but there can only be a limited number of hardware breakpoints as restricted by the CPU hardware. Moreover, ring 0 malware can detect a presence of the hardware breakpoints by accessing the corresponding hardware registers. VMPiRE [24] aims to address the limitations of breakpoints, but it still relies on the OS so is not effective against ring 0 malware. We believe stealthy breakpoint insertion with ring 0 malware is an open problem.

MalT tackles this problem by using performance counters to generate SMIs. Essentially, we compare the EIP of the currently executing instruction with the stored breakpoint address during each cycle. We use 4 bytes to store the breakpoint address and 1 byte for a validity flag. Thus, we need only 5 bytes to store such hardware breakpoints. For each Protected Mode instruction, the SMI handler takes the following steps: (1) Check if the target application is the running thread when the SMI is triggered; (2) check if the current EIP equals a stored breakpoint address; (3) start to count retired instructions in the performance counter, and set the corresponding performance counter to the maximum value; (4) configure LAPIC so that the performance counter overflow generates an SMI.

Breakpoint addresses are stored in SMRAM, and thus the number of active breakpoints we can have is limited by the size of SMRAM. In our system, we reserve a 512-byte region from SMM_BASE+0xFC00 to SMM_BASE+0xFE00. Since each hardware breakpoint takes 5 bytes, we can store a total 102 breakpoints in this region. If necessary, we can expand the total region of SMRAM by taking advantage of a region called `TSeg`, which is configurable via the SMM_MASK register [49]. In contrast to the limited number of hardware breakpoints on the x86 platform, MalT is capable of storing more breakpoints in a more transparent manner.

6.6 Step-by-Step Execution Debugging

As discussed above, we break the execution of a program by using different performance counters. For instance, by monitoring the Retired Instruction event, we can achieve instruction-level stepping in the system. Table 2 summarizes the performance counters we used in our prototype. First, we assign the event to the PerfEvtSel0 register to indicate that the event of interest will be monitored. Next, we set the value of the counter to the maximum value (i.e., a 48-bit register is assigned $2^{48} - 2$). Thus, the next event to increase the value will cause an overflow, triggering an

SMI. Note that the -2 term is used because the Retired Instruction event also counts interrupts. In our case, the SMI itself will cause the counter to increase as well, so we account for that change accordingly. The system becomes deadlocked if the value is not chosen correctly.

Vogl and Eckert [50] also proposed the use of performance counters for instruction-level monitoring. It delivers a Non-Maskable Interrupt (NMI) to force a VM Exit when a performance counter overflows. However, the work is implemented on a hypervisor. MALT leverages SMM and does not employ any virtualization, which provides a more transparent execution environment. In addition, their work [50] incurs a time gap between the occurrence of a performance event and the NMI delivery, while MALT does not encounter this problem. Note that the SMI has priority over an NMI and a maskable interrupt as well. Among these seven stepping methods, instruction-by-instruction stepping achieves fine-grained tracing, but at the cost of a significant performance overhead. Using the Retired Near Returns event causes low system overhead, but it only provides coarse-gained debugging.

6.7 System Restoration

Restoring a system to a clean state after each debugging session is critical to the safety of malware analysis on bare metal. In general, there are two approaches to restore a system: *reboot* and *bootless*. The rebooting approach only needs to reimage the non-volatile devices (e.g., hard disk or BIOS), but it is relatively slow. The bootless approach must manually reinitialize the system state, including memory and disks, but takes less time. For the bootless approach, besides memory and disk restoration, hardware devices also must be restored. Modern I/O devices now have their own processors and memory (e.g., GPU and NIC); quickly and efficiently reinitializing these hardware devices is a challenging problem.

BareBox [14] used a rebootless approach to restore the memory and disk of the analysis machine. However, BareBox only focuses on user-level malware; it disables loading new kernel modules and prevents user-mode access to kernel memory. In other words, ring 0 malware can easily detect the presence of BareBox using a memory scan and manipulate the restoration process, while MALT can successfully operate in the presence of kernel and hypervisor rootkits. Additionally, BareBox does not fully restore the I/O devices (e.g., the internal memory of GPU). BareCloud [29] used LVM-based copy-on-write to restore a remote storage disk. MALT can also use the similar approach to restore the disk. However, this method introduces footprints (e.g., configurations for the remote disk) that malware can detect, which violates the transparency goal.

6.7.1 System Restoring in MALT

To completely restore the debugging server, we consider four components in MALT: (1) volatile memory (i.e., RAM), (2) I/O devices, (3) hard disks, and (4) the BIOS. For the first and second components, MALT uses the reboot approach to restore them. Since we reboot the debugging server, the memory and I/O devices are reset to clean states. This addresses the problem of system restore for malware analysis—I/O devices and kernel memory reinitialization when ring 0 malware exists.

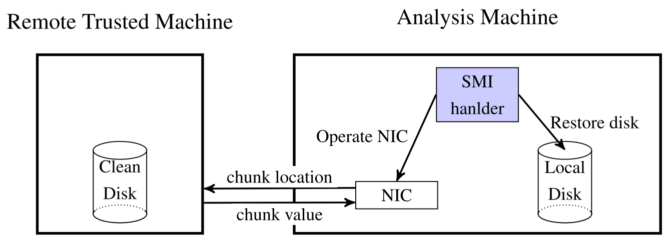


Fig. 2. Architecture of disk restoring in MALT.

For the third component, we reimage the disk by using SMM. Since the debugged malware has ring 0 privilege in our threat model, we cannot use the disk restoration tools in OSes or hypervisors. One simple solution is to take out the disk and restore it in another machine. However, this is not convenient for users. In MALT, we use SMM, as a trusted execution environment, to remotely reimage the disk over the network. Fig. 2 shows the architecture of disk restoring in MALT. The analysis machine is the Debugging Server that runs malware; the remote trusted machine stores a copy of the clean disk and we re-use the Debugging Client as the remote trusted machine. We use the SMI handler to copy the clean image from the remote trusted machine to the analysis machine. Network packets are used for their communication.

Since we do not trust any code in the OS including device drivers, we write two simple device drivers in the SMI handler. One is for the hard disk, and the other one is for the network card (i.e., NIC). We use Serial ATA (SATA) to connect the Southbridge of the motherboard to the hard disk. The I/O base address of the primary SATA controller is at `0x38a0` in the testbed. We can access disk data by performing SATA read/write operations. Note that the disk access is at the block-level, and the sector size is 512-byte. For the network card, the BIOS and OS have initialized it when booting, so we only need to write the transmit/receive descriptors to enable network communication.

Since replacing the whole disk image is time consuming (based on our experiment, it takes about 8 hours to replace a 500 GB disk sector-by-sector), we only restore the modified contents on the disk. In this case, we use a bitmap to record the modified sectors. If we use a single bit for a sector, the bitmap requires 128 MB for a 500 GB disk with a sector size of 512-byte. To reduce the size of the bitmap, we group sectors into a 32 KB chunk. This method is similar to BareBox [14]. To record the modified chunks, we trigger an SMI for each SATA operation. Specifically, we configure the IRQ 14 in I/O APIC to reroute a SATA controller interrupt to become an SMI. Next, we check the SATA operation to see if it is a write. If it is, we mark the bitmap at corresponding location, and the bitmap is stored in the trusted SMRAM.

After a debugging session, the SMI handler restores the disk by chunks. It looks the bitmap and sends packets to the remote trusted machine to retrieve the value of the modified chunks. We repeat this process until all chunks on the disk have been restored. Note that MALT uses SMM to restore the disk at the block-level, so this approach is file-system friendly and OS-agnostic.

As for the last component, the BIOS, we use a tool Flashrom [51] to flash firmware. We need to flash the BIOS twice for a debugging session; once after the debugging session to inject the modified SMM code into the BIOS; and once

TABLE 3
Summary of Anti-Debugging, Anti-VM, and Anti-Emulation Techniques

Anti-debugging [8], [9]	
API Call	Kernel32!IsDebuggerPresent returns 1 if a target process is being debugged ntdll!NtQueryInformationProcess: ProcessInformation field set to -1 if the process is being debugged kernel32!CheckRemoteDebuggerPresent returns 1 in debugger process NtSetInformationThread with ThreadInformationClass set to 0x11 will detach some debuggers kernel32!DebugActiveProcess to prevent other debuggers from attaching to a process
PEB Field	PEB!IsDebugged is set by the system when a process is debugged PEB!NtGlobalFlags is set if the process was created by a debugger
Detection	ForceFlag field in heap header (+0x10) can be used to detect some debuggers UnhandledExceptionFilter calls a user-defined filter function, but terminates in a debugging process TEB of a debugged process contains a NULL pointer if no debugger is attached; valid pointer if some debuggers are attached Ctrl-C raises an exception in a debugged process, but the signal handler is called without debugging Inserting a Rogue INT3 opcode can masquerade as breakpoints Trap flag register manipulation to thwart tracers If entryPoint RVA is set to 0, the magic MZ value in PE files is erased ZwClose system call with invalid parameters can raise an exception in an attached debugger Direct context modification to confuse a debugger 0x2D interrupt causes debugged program to stop raising exceptions Some In-circuit Emulators (ICEs) can be detected by observing the behavior of the undocumented 0xF1 instruction Searching for 0xCC instructions in program memory to detect software breakpoints TLS-callback to perform checks
Anti-virtualization	
VMWare	Virtualized device identifiers contain well-known strings [7] *checkvm* software [54] can search for VMWare hooks in memory Well-known locations/strings associated with VMWare tools
Xen	Checking the VMX bit by executing CPUID with EAX as 1 [28] CPU errata: AH4 erratum [28]
Other	LDTR register [10] IDTR register (Red Pill [27]) Magic I/O port (0x5658, 'VX') [14] Invalid instruction behavior [11] Using memory deduplication to detect various hypervisors including VMware ESX server, Xen, and Linux KVM [55]
Anti-emulation	
Bochs	Visible debug port [7]
QEMU	`cpuid` returns less specific information [4] Accessing reserved MSR registers raises a General Protection (GP) exception in real hardware; QEMU does not [12] Attempting to execute an instruction longer than 15 bytes raises a GP exception in real hardware; QEMU does not [12] Undocumented `icebp` instruction hangs in QEMU [4], while real hardware raises an exception Unaligned memory references raise exceptions in real hardware; unsupported by QEMU [12] Bit 3 of FPU Control World register is always 1 in real hardware, while QEMU contains a 0 [4]
Other	Using CPU bugs or errata to create CPU fingerprints via public chipset documentation [12]

before the subsequent debugging session to remove the footprint of the BIOS. Section 7 details this.

7 Transparency Analysis

In this paper, we consider the transparency of four subjects. They are (1) virtualization, (2) emulation, (3) SMM, and (4) debuggers. Next, we discuss the transparency of these subjects one by one.

Virtualization. The transparency of virtualization is difficult to achieve. For instance, Red Pill [27] uses an unprivileged instruction `SIDT` to read the interrupt descriptor (IDT) register to determine the presence of a virtual machine. To work on multi-processor system, Red Pill needs to use `SetThreadAffinityMask()` Windows API call to limit thread execution to one processor [10]. nEther [28] detects hardware virtualization using CPU design defects. Furthermore, there are many footprints introduced by virtualization such as well-known strings in memory [7], magic I/O ports [14], and invalid instruction behaviors [11]. Moreover, Garfinkel et al. [13] argued that building a transparent virtual machine is impractical.

Emulation. Researchers have used emulation to debug malware. QEMU simulates all the hardware devices including CPU, and malware runs on top of the emulated software. Because of the emulated environment, malware can detect it. For example, accessing a reserved or unimplemented MSR register causes a general protection exception, while QEMU does not raise an exception [12]. The underlying problem is the design of emulator does not have transparent malware analysis in mind (e.g., emulator architect may not implement CPU errata). Table 3 shows more anti-emulation techniques. In theory, these defects could be fixed, but it is impractical to patch all of them in a timely manner.

SMM. As explained in Section 2, SMM is a hardware feature existing in all x86 machines. Regarding its transparency, the Intel manual [21] specifies the following mechanisms that make SMM transparent to the application programs and operating systems: (1) the only way to enter SMM is by means of an SMI; (2) the processor executes SMM code in a separate address space (SMRAM) that is inaccessible from the other operating modes; (3) upon entering SMM, the processor saves the context of the interrupted program or task; (4) all interrupts normally handled by the operating system are disabled upon entry into SMM; and (5) the `RSM` instruction can be executed only in SMM. Note that SMM steals CPU time from the running program, which is a side effect of SMM. For instance, malware can detect SMM based on the time delay. Even so, SMM is still more transparent than virtualization and emulation.

Debuggers. An array of debuggers have been proposed for transparent debugging. These include in-guest [24], [25], emulation-based [5], [26], and virtualization-based [1], [2] approaches. MALT is an SMM-based system. As to the transparency, we only consider the artifacts introduced by debuggers themselves, not the environments (e.g., hypervisor or SMM). Ether [1] proposes five formal requirements for achieving transparency, including (1) high privilege, (2) no non-privileged side effects, (3) identical basic instruction execution semantics, (4) transparent exception handling, and (5) identical measurement of time. MALT satisfies the first requirement by running the analysis code in SMM with ring -2. We enumerate all the side effects introduced by MALT in Section 7.1 and attempt to meet the second requirement in our system. Since MALT runs on bare metal, it immediately meets the third and fourth requirements. Last, MALT partially satisfies the fifth requirement by adjusting the local timers in the SMI handler. We further discuss the timing attacks below.

7.1 Side Effects Introduced by MALT

MALT aims to transparently analyze malware with minimum footprints. Here we enumerate the side effects introduced by MALT and show how we mitigate them. Note that achieving the highest level of transparency requires MALT to run in single-stepping mode.

CPU. We implement MALT in SMM, another CPU mode in the x86 architecture, which provides an isolated environment for executing code. After recognizing the SMI assertion, the processor saves almost the entirety of its state to SMRAM. As previously discussed, we rely on the performance monitoring registers and LAPIC to generate SMIs. Although these registers are inaccessible from user-level malware, attackers with ring 0 privilege can read and modify them. LAPIC registers in the CPU are memory-mapped, and its base address is normally at `0xFEE00000`. In MALT, we relocate LAPIC registers to another physical address by modifying the value in the 24-bit base address field of the IA32_APIC_BASE Model Specific Register [21]. To find the LAPIC registers, attackers need to read IA32_APIC_BASE MSR first that we can intercept. Performance monitoring registers are also MSRs. `RDMSR`, `RDPMC`, and `WRMSR` are the only instructions that can access the performance counters [49] or MSRs. To mitigate the footprints of these MSRs, we run MALT in the instruction-by-instruction mode and adjust the return values seen by these instructions before resuming Protected Mode. If we find a `WRMSR` to modify the performance counters, the debugger client will be notified.

Memory and cache. MALT uses an isolated memory region (SMRAM) from normal memory in Protected Mode. Any access to this memory in other CPU modes will be redirected to VGA memory. Note that this memory redirection occurs in all x86 machines, even without MALT; this is not unique to our system. In 2009, Intel introduced System Management Range Registers (SMRR) [21] that limits cache references of addresses in SMRAM to code running in SMM. This is the vendor's response to the cache poisoning attack [52] and SMRR is available on all Intel processors since then. The AMD64 architecture does not have SMRR, but the processor internally keeps track of SMRAM and system memory accesses separately and properly handles situations where aliasing occurs (i.e., main-memory locations as aliases for SMRAM-memory locations) [53]. MALT does not flush the cache when entering and exiting SMM to avoid cache-based side-channel detection.

IO configurations and BIOS. MALT reroutes a serial interrupt to generate an SMI to initialize a debugging session, and the modified redirection table entry in I/O APIC can be read by malware with ring 0 privilege. We change the redirection table entry back to its original value to remove this footprint in the first generated SMI handler. Once SMM has control of the system, the SMIs are triggered by configuring performance counters. MALT uses a custom BIOS, Coreboot, to program the SMM code. An attacker with ring 0 privilege can check the hash value of the BIOS to detect the presence of our system. To avoid this fingerprint, we flash the BIOS with the original image before the debugging process using the tool Flashrom [51], and it flashes the Coreboot with the original AMI BIOS. At that time, the SMI handler, including the MALT code, has been loaded into SMRAM and locked. Note that we also need to reflash the Coreboot image for the next system restart.

Timing. There are many timers and counters on the motherboard and chipsets, such as the Real Time Clock (RTC), the Programmable Interval Timer (8253/8254 chip), the High Precision Event Timer (HPET), the ACPI Power Management Timer, the APIC Timer, and the Time Stamp Counter (TSC). Malware can read a timer and calculate its running time. If the time exceeds a certain threshold, malware can conclude that a debugger is present. For the configurable timers, we record their values after switching into SMM. When SMM exits, we set the values back using the recorded values minus the SMM switching time. Thus, malware is unaware of the time spent in the SMI handler. However, some of the timers and counters cannot be changed, even in SMM. To address this problem, we adjust the return values of these timers in the instruction-level stepping mode. For example, the `RDTSC` instruction reads the TSC register and writes the value to the `EAX` and `EDX` registers. While debugging, we can check if the current instruction is `RDTSC` and adjust the values of `EAX` and `EDX` before leaving the SMI handler.

Unfortunately, MALT cannot defend against timing attacks involving an external timer. For instance, malware can send a packet to a remote server to get correct timing information (e.g., NTP service). In this case, malware can detect the presence of our system and alter its behavior accordingly. One potential solution to address this problem is to intercept the instruction that reaches out for timing information and prepare a fake time for the OS. Naturally, this would not be foolproof as an attacker could retrieve an encrypted time from a remote location. Such attacks are difficult to contend with because we cannot always know when a particular packet contains timing information. To the best of our knowledge, all existing debugging systems with any measurable performance slowdown suffer from

TABLE 4
Running Packed `Notepad.exe` under Different Environments

Packing Tool	MALT	OllyDbg V1.10	DynamoRIO V4.2.0-3	VMware Fusion V6.0.2
UPX V3.08	OK	OK	OK	OK
Obsidium V1.4	OK	Access violation at 0x00000000	Segmentation fault	OK
ASPack V2.29	OK	OK	OK	OK
Armadillo V2.01	OK	Access violation at 0x42434847	Crash	Crash
Themida V2.2.3.0	OK	Privileged instruction exception	Exception at 0x10a65d7	Message: cannot run under a VM
RLPack V1.21	OK	OK	OK	OK
PELock V1.0694	OK	Display message and terminate	Segmentation fault	OK
VMProtect V2.13.5	OK	Message: a debugger was found	OK	Crash
eXPressor V1.8.0.1	OK	Message: unknown executable format	Segmentation fault	Crash
PECompact V3.02.2	OK	Access violation at 0x00000000	OK	OK

this attack. However, external timing attacks require network communications and thus dramatically increase the probability that the malware will be flagged. We believe that malware will avoid using external timing attacks precisely because it wants to minimize its footprint on the victim's computer, including using spin loops. We can also analyze portions of the malware separately and amortize the analysis time.

7.2 Analysis of Anti-Debugging, -VM, and -Emulation Techniques

To analyze the transparency of MALT system, we employ anti-debugging, anti-virtualization and anti-emulation techniques from [7], [9], [10], [11], [12] to verify our system. Since MALT runs on a bare-metal machine, these anti-virtualization techniques will no longer work on it. Additionally, MALT does not change any code or the running environments of operating systems and applications so that normal anti-debugging techniques cannot work against it. For example, the debug flag in the `PEB` structure on Windows will not be set while MALT is running. Table 3 summarizes popular anti-debugging, anti-virtualization, and anti-emulation techniques, and we have verified that MALT can evade all these detection techniques.

7.3 Testing with Packers

Packing is used to obfuscate the binary code of a program. It is typically used to protect the executable from reverse-engineering. Nowadays, malware writers also use packing tools to obfuscate their malware. Packed malware is more difficult for security researchers to reverse-engineer the binary code. In addition, many packers contain anti-debugging and anti-VM features, further increasing the challenge of reverse-engineering packed malware.

To demonstrate the transparency of MALT, we use popular packing tools to pack the `Notepad.exe` application in a Windows environment and run this packed application in MALT with near return stepping mode, OllyDbg [56], DynamoRIO [57], and a Windows XP SP3 in VMware Fusion [58], respectively. Ten packing tools are used, including UPX, Obsidium, ASPack, Armadillo, Themida, RLPack, PELock, VMProtect, eXPressor, and PECompact. All these packing tools enable the settings for anti-debugging and anti-VM functions if they have them. After running the packed `Notepad.exe`, if the Notepad window appears, we know that it has launched successfully. Table 4 lists the results. All the packing tools except UPX, ASPack, and RLPack can detect OllyDbg. Obsidium, Armadillo, Themida, PELock, and eXPressor are able to detect DynamoRIO, and the VM can be detected by Armadillo, Themida, VMProtect, and eXpressor. In contrast, MALT remains transparent to all these packing tools as we expected.

7.4 Transparency of MALT

Functions and code added by MALT. Sections 7.2 and 7.3 show that existing anti-debugging, anti-VM, anti-emulation, and packing techniques cannot detect the presence of MALT. This is because the current techniques are not targeting MALT's functions or code, so it is possible that future malware could detect MALT due to the ever-present cat-and-mouse game between attackers and defenders. As for 'tomorrow's malware,' we enumerate and mitigate the side effects introduced by MALT in Section 7.1. Note that mitigating all footprints requires a high level of granularity provided by stepping instructions. As with other debugging systems, MALT cannot defend against external timing attacks.

Running environment used by MALT. MALT is built on SMM so that the transparency of MALT depends on the implications of SMM usage. Since SMM is not intended for debugging, the hardware and software on the system may not expect this usage, which may introduce side-channel footprints for attackers to detect MALT (e.g., performance slowdown and frequent switching). However, we believe using SMM is more transparent than using virtualization or emulation as done in previous systems due to its minimal TCB and attack surface.

Towards true transparency. Debugging transparency is a challenging and recently active problem in the security community. Unlike previous solutions that use virtualization or emulation, MALT isolates the execution in the CPU, which provides a novel idea of addressing the transparency problem. Although MALT is not fully transparent, we would like to draw the attention of the community to this hardware-based approach because the running environment of the debugger is more transparent than those of previous systems (i.e., virtualization and emulation). Moreover, we further argue hardware support for truly transparent debugging. For instance, there could be a dedicated and well-designed CPU mode for debugging, perhaps with performance counters that are inaccessible from other CPU modes, which provides a transparent switching method between CPU modes.

8 Evaluation

8.1 Testbed Specification and Code Size

We evaluate MALT using two physical machines. The target server used an ASUS M2V-MX_SE motherboard with an AMD K8 northbridge and a VIA VT8237r southbridge. It has a 2.2 GHz AMD LE-1250 CPU and 2GB Kingston DDR2 RAM. The target machine uses Windows XP SP3, CentOS 5.5 with kernel 2.6.24, and Xen 3.1.2 with CentOS 5.5 as domain 0. To simplify the installation, they are installed on three separate hard disks, and the SeaBIOS manages the boot process. The hard disks we used are Seagate Barracuda 7,200 RPM with a 500 GB capacity. The debugging client is a Dell Inspiron 15R laptop with Ubuntu 12.04 LTS. It uses a 2.4 GHz Intel Core i5-2430M CPU and 6 GB DDR3 RAM. We use a USB 2.0 to Serial (9-Pin) DB-9 RS-232 converter cable to connect two machines for debugging commands.

We use `cloc` [59] to compute the number of lines of source code. Coreboot and its SeaBIOS payload contain 248,421 lines. MALT adds about 1,900 lines of C code in the SMI hander. After compiling the Coreboot code, the size of the image is 1 MB, and the SMI hander contains 3,749 bytes. The debugger client contains 494 lines of C code.

8.2 Breakdown of Operations in MALT

To understand the performance of our debugging system, we measure the time elapsed during particular operations in the SMI handler. We use the Time Stamp Counter to measure the number of CPU cycles elapsed during each operation; we multiplied the clock frequency by the delta in TSCs. After a performance counter triggers an SMI, the system hardware automatically saves the current architectural state into SMRAM and begins executing the SMI handler. The first operation in the SMI handler is to identify the last running process in the CPU. If the last running process is not the target malware, we only need to configure the performance counter register for the next SMI and exit from SMM. Otherwise, we perform several checks. First, we check for newly received messages and whether a breakpoint has been reached. If there are no new commands and no breakpoints to evaluate, we reconfigure the performance counter registers for the next SMI. Table 5 shows a breakdown of the operations in the SMI handler if the last running process is the target malware. This experiment shows the mean, standard deviation, and 95 percent confidence interval of 25 runs. The SMM switching time takes about 3.29 microseconds. Command checking and breakpoint checking take about 2.19 microseconds in total. Configuring performance monitoring registers and SMI status registers for subsequent SMI generation takes about 1.66 microseconds. Last, SMM resume takes 4.58 microseconds. Thus, when the last process is the target malware, MALT takes about 12 microseconds to execute an instruction without debugging command communication.

TABLE 5
Breakdown of SMI Handler (Time: μs)

Operations	Mean	STD	95% CI
SMM switching	3.29	0.08	[3.27,3.32]
Command and BP checking	2.19	0.09	[2.15,2.22]
Next SMI configuration	1.66	0.06	[1.64,1.69]
SMM resume	4.58	0.10	[4.55,4.61]
Total	11.72		

8.3 Step-by-Step Debugging Overhead

To demonstrate the efficiency of our system, we measure the performance overhead of the seven stepping methods on both Windows and Linux platforms. We use a benchmark program, SuperPI [60], version 1.8 on Windows and version 2.0 on Linux. SuperPI is a single-threaded benchmark that calculates the value of π to a specific number of digits and outputs the calculation time. This tightly written, arithmetic-intensive benchmark is suitable for evaluating CPU performance. SuperPI calculates 64 K digits of π, and it takes 1.610 and 1.898 s on Windows and Linux, respectively. Note that the speed of π calculation varies, depending on the selected algorithm. Additionally, we use a popular Linux Command, `gzip`, to compress 4M digits of π to measure the performance overhead. The 4M digits of π is generated by the SuperPI program. On Windows, we install `Cygwin` to execute `gzip` version 1.4. On Linux, we use `gzip` version 1.3.5. The compression operation takes 1.875 s on Windows and 1.704 s on Linux. After we run the programs without MALT, we enable each of the seven stepping methods separately and record the runtimes. SuperPI shows the runtimes, and we use shell scripts to calculate the runtimes of the `gzip` command. We run each experiment five times and show the average results in Table 6.

Table 6 shows the performance slowdown introduced by the step-by-step debugging. The first column specifies different stepping methods; the following four columns show the

TABLE 6
Stepping Overhead on Windows and Linux

Stepping methods	Runtime (Seconds)				Slowdown			
	Windowx		Linux		Windowx		Linux	
	π	*gzip*	π	*gzip*	π	*gzip*	π	*gzip*
Without MALT	1.610 s	1.875 s	1.898 s	1.704 s	1.00x	1.00x	1.00x	1.00x
Retired far control transfers	2.230 s	2.564 s	2.495 s	2.432 s	1.38x	1.36x	1.46x	1.42x
Retired near returns	74.43 s	73.14 s	61.56 s	59.08 s	46.2x	39.1x	36.1x	34.7x
Retired taken branches mispredicted	155.3 s	145.7 s	68.42 s	138.3 s	96.5x	40.2x	77.7x	81.2x
Retired taken branches	1020 s	1754 s	476.6 s	1538 s	634x	935x	280x	903x
Retired mispredicted branches	160.3 s	280.0 s	77.31 s	236.3 s	99.6x	149x	45.4x	138x
Retired branches	1200 s	2243 s	494 s	1760 s	745x	1196x	290x	1033x
Retired instructions	1645 s	2849 s	839 s	2333 s	1021x	1519x	492x	1369x

running time of the SuperPI and `gzip`; and the last four columns represent the slowdown of the programs, which is calculated by dividing the current running time by the base running time. It is evident that far control transfer (e.g., IRET instruction) stepping only introduces a 1.5× slowdown on Windows and Linux, which facilitates coarse-grained tracing for malware debugging. As expected, fine-grained stepping methods introduce more overhead. The instruction-by-instruction debugging causes about 1519× slowdown on Windows for `gzip` compressing 4M digits of π, which demonstrates the worst-case performance degradation in our debugging methods. One way to improve the performance is to reduce the time used for SMM switching and resume operations by cooperating with hardware vendors. Note that MALT is twice as fast as Ether [1], [4] in the single-stepping mode. Table 5 shows that MALT takes 12 microseconds to execute an instruction if the last running process is the target malware (i.e., if MALT performs an analysis). Table 6 shows the system overhead incurred by each stepping mode when no target malware is executing (i.e., when MALT immediately returns from SMM). By analyzing the results in Table 6, we observe that the time per instruction is less than observed in Table 5. We believe this is due to hardware caching. The CPU saves the current architectural state into SMRAM when switching into SMM. If the system keeps switching for each instruction, hardware may not write the same contents back into memory due to hardware caching.

Despite a three order-of-magnitude slowdown on Windows, the debugging target machine is still usable and responsive to user interaction. In particular, the instruction-by-instruction debugging is intended for use by a human operator from the client machine, and we argue that the user would not notice this overhead while entering the debugging commands (e.g., `Read Register`) on the client machine.[1] We believe that achieving high transparency at the cost of performance degradation is necessary for certain types of malware analysis. Note that the overhead in Windows is larger than that in Linux. This is because (1) the semantic gap problem is solved differently in each platform, and (2) the implementations of the benchmark programs are different.

8.4 System Restoration Overhead

To measure the overhead of a complete system restoration in MALT, we measure the time taken by each restoration step including system rebooting, disk restoration, and BIOS flashing. To calculate the time taken to reboot the system, we start an external timer when the OS executes the reboot command (i.e., `reboot` on Linux and `shutdown -h now` on Windows), and we stop the timer when the OS GUI is displayed after rebooting. Note that the booting time is OS-dependent, and we use Linux in the experiment. Since we only restore the changed chunks in the disk, the time taken by disk restoration depends on the number of the modified chunks. In the experiment, we make a copy of a file with size of 10 MB and save it on the disk. Then, we use SMM to restore the disk. We use the TSC to measure the time elapsed for disk restoration and flashing the BIOS. Table 7 shows the breakdown of the system restoration process. System rebooting takes about 25 seconds. This includes OS shutdown, BIOS initialization, boot loader execution, and OS loading. Since we only need to restore the modified contents in the disk (i.e., 10 MB file and OS system logs), the hard disk restoration only takes 21 seconds. For the BIOS flashing, we need to flash the BIOS twice; one is to flash Coreboot in the BIOS before rebooting, and the other is to flash original BIOS back after rebooting to remove fingerprints. Each flashing takes about 28 seconds, yielding a total of about 56 seconds to flash the BIOS twice. Therefore, the total time for system restoration process is about 102 seconds. Compared to BareBox [14], MALT takes longer time to complete system restoration. However, BareBox relies on a Meta-OS and only works for user-level malware, while MALT is capable of analyzing ring 0 code so that privileged malware cannot tamper the restoration process. We believe our system provides a more reliable approach for system restoration in malware analysis.

TABLE 7
Breakdown of System Restoration Process (Time: s)

Steps	Mean	STD	95% CI
System rebooting	25.03	1.01	[24.01, 26.12]
Hard disk restoration	20.75	2.33	[17.39, 22.34]
BIOS flashing	56.23	1.34	[54.55, 57.97]
Total	102.01		

9 Discussion and Limitations

MALT uses SMM as the foundation to implement various debugging functions. Before 2006, computers did not lock their SMRAM in the BIOS [40], and researchers used this flaw to implement SMM-based rootkits [39], [40], [41]. Modern computers lock the SMRAM in the BIOS so that SMRAM is inaccessible from any other CPU modes after booting. Wojtczuk and Rutkowska demonstrated bypassing the SMRAM lock through memory reclaiming [33] or cache poisoning [52]. The memory reclaiming attack is addressed by locking the remapping registers and Top of Low Usable DRAM (TOLUD) register. The cache poisoning attack forces the CPU to execute instructions from the cache instead of SMRAM by manipulating the Memory Type Range Register (MTRR). Duflot also independently discovered this architectural vulnerability [61], but it has been fixed by Intel adding SMRR [21]. Furthermore, Duflot et al. [62] listed some design issues of SMM, but they can be fixed by correct configurations in BIOS and careful implementation of the SMI handler. Wojtczuk and Kallenberg [63] recently presented an SMM attack by manipulating UEFI boot script that allows attackers to bypass the SMM lock and modify the SMI handler with ring 0 privilege. Fortunately, as stated in the paper [63], the BIOS update around the end of 2014 fixed this vulnerability. In MALT, we assume that SMM is trusted.

Butterworth et al. [64] demonstrated a buffer overflow vulnerability in the BIOS updating process in SMM, but this is not an architectural vulnerability and is specific to that particular BIOS version. (Our SMM code does not contain that vulnerable code). Since MALT adds 1,500 lines of C

1. To visualize the performance slowdown, we record a video (https://youtu.be/NP6Bb4CdqN0) that shows MALT operating in the instruction-stepping mode in Windows (cf. highest overhead in Table 6).

code in the SMI handler, it is possible that our code has bugs that could be exploited. Fortunately, SMM provides a strong isolation from other CPU modes (i.e., it has its own sealed memory). The only inputs from a user are through serial messages, making it difficult for malicious code to be injected into our system. We implement MALT on a single-core processor for compatibility with Coreboot, but SMM also works on multi-core systems [21]. Each core has its own set of MSR registers, which define the SMRAM region. When an SMI is generated, all the cores will enter into SMM with their own SMI handler. One simple way is to let one core execute our debugging code and spin the other cores until the first has finished. SMM-based systems such as HyperSentry [35] and SICE [65] are implemented on multi-core processors. In a multi-core system, MALT can debug a process by pinning it to a specific core while allowing the other cores to execute the rest of the system normally. This will change thread scheduling for the debugged process by effectively serializing its threads which may be detectable by an adversary. Recently, Intel introduced SMM-Transfer Monitor (STM), which virtualizes the SMM code [21]. It is also the answer to attacks against Trust Execution Technology (TXT) [66]. Unfortunately, the use of an STM involves blocking SMIs, which potentially prevents our system from executing. However, we can modify the STM code in SMRAM, which executes in SMM, to provide the functionality without affecting our system.

System Management Mode exists in all current x86 devices. There is no indication that Intel will remove SMM from the x86 architecture. Considering the popularity of SMM in computing systems, we believe SMM-based research is still important and valuable. Although SMM is not designed for debugging, SMM-like capabilities could be leveraged to aid transparent debugging. In fact, SMM is a mechanism that essentially provides an isolated computing fabric and the hardware support for meeting MALT's needs. We would like to emphasize this as an architectural principle for debugging. Our prototype leverages the isolation principles currently provided by SMM, but this does not mean that the MALT architecture must use SMM; rather, it is merely a mechanism that implements the required security policies for MALT. We would further argue for desirability of architectural support in aiding debugging transparency.

10 CONCLUSIONS

In this paper, we developed MALT, a bare-metal debugging system that employs System Management Mode to transparently analyze armored malware. As a hardware-assisted debugging system, MALT has a smaller TCB than hypervisor-based approaches. Moreover, it is immune to hypervisor attacks and is capable of analyzing and debugging hypervisor-based rootkits and OS kernels. It also introduces minimum artifacts while achieving transparency. Through extensive experiments, we have demonstrated that MALT remains transparent in the presence of all tested packers, anti-debugging, anti-virtualization, and anti-emulation techniques. Moreover, MALT provides a novel technique that completely and reliably restores the target system to a clean state. MALT introduces moderate but manageable overheads on Windows and Linux, which range from 1.46 to 1519 times slowdown, depending on the stepping method. The complete restoration of a system takes about 2 minutes.

REFERENCES

[1] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware analysis via hardware virtualization extensions," in *Proc. 15th ACM Conf. Comput. Commun. Security*, 2008, pp. 51–62.

[2] Z. Deng, X. Zhang, and D. Xu, "SPIDER: Stealthy binary program instrumentation and debugging via hardware virtualization," in *Proc. Annu. Comput. Security Appl. Conf.*, 2013, pp. 289–298.

[3] A. Fattori, R. Paleari, L. Martignoni, and M. Monga, "Dynamic and transparent analysis of commodity production systems," in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2010, pp. 417–426.

[4] L.-K. Yan, M. Jayachandra, M. Zhang, and H. Yin. (2012). V2E: Combining hardware virtualization and software emulation for transparent and extensible malware analysis. in *Proc. 8th ACM SIGPLAN/SIGOPS Conf. Virtual Execution Environ.* [Online]. Available: http://doi.acm.org/10.1145/2151024.2151053

[5] (2009). Anubis. Analyzing Unknown Binaries [Online]. Available: http://anubis.iseclab.org

[6] N. A. Quynh and K. Suzaki. (2010). "Virt-ICE: Next-generation debugger for malware analysis," in *Black Hat USA*. [Online]. Available: https://media.blackhat.com/bh-us-10/whitepapers/Anh/BlackHat-USA-2010-Anh-Virt-ICE-wp.pdf

[7] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario, "Towards an understanding of Anti-Virtualization and Anti-Debugging behavior in modern malware," in *Proc. 38th Annu. IEEE Int. Conf. Dependable Syst. Netw.*, 2008, pp. 177–186.

[8] R. R. Branco, G. N. Barbosa, and P. D. Neto. (2012). "Scientific but not academical overview of malware anti-debugging, anti-disassembly and Anti-VM technologies," in *Black Hat* [Online]. Available: https://www.blackhat.com/docs/us-14/materials/us-14-Branco-Prevalent-Characteristics-In-Modern-Malware.pdf

[9] N. Falliere. (2010). Windows anti-debug reference [Online]. Available: http://www.symantec.com/connect/articles/windows-anti-debug-reference

[10] D. Quist and V. Val Smith. (2006). Detecting the presence of virtual machines using the local data table [Online]. Available: http://www.offensivecomputing.net

[11] E. Bachaalany. (2005). Detect if your program is running inside a virtual machine [Online]. Available: http://www.codeproject.com/Articles/9823/Detect-if-your-program-is-running-inside-a-Virtual

[12] T. Raffetseder, C. Kruegel, and E. Kirda, "Detecting system emulators," in *Information Security*. Berlin, Germany: Springer, 2007.

[13] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin, "Compatibility is not transparency: VMM detection myths and realities," in *Proc. 11th USENIX Workshop Hot Topics Operating Syst.*, 2007. pp. 1–6.

[14] D. Kirat, G. Vigna, and C. Kruegel, "BareBox: Efficient malware analysis on Bare-metal," in *Proc. 27th Annu. Comput. Security Appl. Conf.*, 2011, pp. 403–412.

[15] C. Willems, R. Hund, A. Fobian, D. Felsch, T. Holz, and A. Vasudevan, "Down to the bare Metal: Using processor features for binary analysis," in *Proc. Annu. Comput. Security Appl. Conf.*, 2012, pp. 189–198.

[16] K. Kortchinsky. (2009). "CLOUDBURST: A VMware guest to host escape story," in *Black Hat USA* [Online]. Available: https://www.blackhat.com/presentations/bh-usa-09/KORTCHINSKY/BHUSA09-Kortchinsky-Cloudburst-SLIDES.pdf

[17] R. Wojtczuk, J. Rutkowska, and A. Tereshkin. (2008). "Xen 0wning Trilogy," in *Black Hat USA* [Online]. Available: http://invisiblethingslab.com/resources/bh08/part3.pdf

[18] S. T. King and P. M. Chen, "SubVirt: Implementing malware with virtual machines," in *Proc. 27th IEEE Symp. Security Privacy*, May 2006, pp. 314–327.

[19] J. Rutkowska. (2006). Blue Pill [Online]. Available: http://theinvisiblethings.blogspot.com/2006/06/introducing-blue-pill.html

[20] F. Zhang, K. Leach, A. Stavrou, H. Wang, and K. Sun, "Using hardware features for increased debugging Transparency," in *Proc. 36th IEEE Symp. Security Privacy*, May 2015, pp. 55–69.

[21] (2015). Intel. 64 and IA-32 architectures software developer's manual [Online]. Available: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

[22] (2015, Jul.). Coreboot. Open-Source BIOS [Online]. Available: http://www.coreboot.org/

[23] (2015, Jul.). SeaBIOS [Online]. Available: http://www.coreboot.org/SeaBIOS

[24] A. Vasudevan and R. Yerraballi, “Stealth breakpoints,” in *Proc. 21st Annu. Comput. Security Appl. Conf.*, 2005, pp. 392–402.

[25] (2015). IDA Pro [Online]. Available: www.hex-rays.com/products/ida/

[26] 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 *Proc. 4th Int. Conf. Inform. Syst. Security*, 2008, pp. 1–25.

[27] J. Rutkowska. (2006). Red Pill [Online]. Available: http://www.ouah.org/Red_Pill.html

[28] G. Pek, B. Bencsath, and L. Buttyan, “nEther: In-guest detection of Out-of-the-guest malware analyzers,” in *Proc. 4th Eur. Workshop Syst. Security*, 2011, pp. 1–6.

[29] D. Kirat, G. Vigna, and C. Kruegel, “BareCloud: Bare-metal Analysis-based evasive malware detection,” in *Proc. 23rd USENIX Security Symp.*, 2014, pp. 287–301.

[30] (2015, Jul.). Ohloh. Black Duck Software, Inc [Online]. Available: http://www.ohloh.net

[31] (2015, Jul.). VMware, Inc. VMWare Workstation [Online]. Available: https://www.vmware.com/products/workstation

[32] K. Kourai, “Fast and correct performance recovery of operating systems using a virtual machine monitor,” in *Proc. 7th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ.*, 2011, pp. 99–110.

[33] J. Rutkowska and R. Wojtczuk. (2008). Preventing and detecting Xen Hypervisor subversions [Online]. Available: http://www.invisiblethingslab.com/resources/bh08/part2-full.pdf

[34] F. Zhang, J. Wang, K. Sun, and A. Stavrou, “HyperCheck: A Hardware-assisted integrity monitor,” *IEEE Trans. Dependable Secure Comput.*, vol. 11, no. 4, pp. 332–344, Jul./Aug. 2014.

[35] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky, “HyperSentry: Enabling stealthy in-context measurement of hypervisor integrity,” in *Proc. 17th ACM Conf. Comput. Commun. Security*, 2010, pp. 38–49.

[36] F. Zhang, K. Leach, K. Sun, and A. Stavrou, “SPECTRE: A dependable introspection framework via system management mode,” in *Proc. 43rd Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2013, pp. 1–12.

[37] J. Wang, F. Zhang, K. Sun, and A. Stavrou, “Firmware-assisted memory acquisition and analysis tools for digital forensic,” in *Proc. 6th Int. Workshop Systematic Approaches Digital Forensic Eng.*, 2011, pp. 1–5.

[38] A. Reina, A. Fattori, F. Pagani, L. Cavallaro, and D. Bruschi, “When hardware meets Software: A bulletproof solution to forensic memory acquisition,” in *Proc. Annu. Comput. Security Appl. Conf.*, 2012, pp. 79–88.

[39] L. Duflot, D. Etiemble, and O. Grumelard, “Using CPU system management mode to circumvent operating system security functions,” in *Proc. 7th CanSecWest Conf.*, 2004, pp. 1–15.

[40] S. Embleton, S. Sparks, and C. Zou, “SMM rootkits: A new breed of OS independent malware,” in *Proc. 4th Int. Conf. Security Privacy Commun. Netw.*, 2008, pp. 11:1–11:12.

[41] BSDaemon, coideloko, and D0nAnd0n. (2008). “System management mode Hack: Using SMM for ‘other purposes’,” *Phrack Mag.* [Online]. Available: http://phrack.org/issues/65/7.html

[42] (2013). NSA’s ANT Division Catalog of Exploits for Nearly Every Major Software/Hardware/Firmware [Online]. Available: http://Leaksource.wordpress.com

[43] Trusted Computing Group. (2012, Feb.). TCG PC client specific implementation specification for conventional BIOS, specification version 1.21 [Online]. Available: http://www.trustedcomputinggroup.org

[44] F. Zhang, H. Wang, K. Leach, and A. Stavrou, “A framework to secure peripherals at runtime,” in *Proc. 19th Eur. Symp. Res. Comput. Security.*, 2014, pp. 219–238.

[45] B. Jain, M. B. Baig, D. Zhang, D. E. Porter, and R. Sion, “SoK: Introspections on trust and the semantic gap,” in *Proc. 35th IEEE Symp. Security Privacy*, 2014, pp. 605–620.

[46] T. Garfinkel and M. Rosenblum, “A virtual machine introspection based architecture for intrusion detection,” in *Proc. 10th Annu. Netw. Distrib. Syst. Security Symp.*, 2003, pp. 191–206.

[47] X. Jiang, X. Wang, and D. Xu, “Stealthy malware detection through VMM-based Out-of-the-box semantic view reconstruction,” in *Proc. 14th ACM Conf. Comput. Commun. Security*, 2007, pp. 128–138.

[48] VIA Technologies, Inc., “VT8237R South Bridge, Revision 2.06,” December 2005.

[49] (2006). Advanced Micro Devices, Inc.. BIOS and Kernel Developer’s Guide for AMD Athlon 64 and AMD Opteron Processors [Online]. Available: http://support.amd.com/us/ProcessorTechDocs/26094.PDF

[50] S. Vogl and C. Eckert, “Using hardware performance events for instruction-level monitoring on the x86 architecture,” in *Proc. Eur. Workshop Syst. Security*, 2012.

[51] (2015, Jul.). Flashrom. Firmware Flash Utility [Online]. Available: http://www.flashrom.org/

[52] R. Wojtczuk and J. Rutkowska. (2009). Attacking SMM memory via intel CPU cache poisoning [Online]. Available: http://invisiblethingslab.com/resources/misc09/smm_cache_fun.pdf

[53] Advanced Micro Devices, Inc.. (2015, Jun.). AMD64 Architecture-Programmer Manual Volume 2: System Programming [Online]. Available: http://support.amd.com/TechDocs/24593.pdf

[54] (2013). checkvm: Scoopy doo [Online]. Available: http://www.trapkit.de/research/vmm/scoopydoo/scoopy_doo.htm

[55] J. Xiao, Z. Xu, H. Huang, and H. Wang, “Security implications of memory deduplication in a virtualized environment,” in *Proc. 43rd Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2013, pp. 1–12.

[56] (2013). OllyDbg [Online]. Available: www.ollydbg.de

[57] (2015, Jul.). DynamoRIO. Dynamic Instrumentation Tool Platform [Online]. Available: http://dynamorio.org/

[58] (2015, Jul.). VMware, Inc. VMWare Fusion [Online]. Available: https://www.vmware.com/products/fusion

[59] (2015, Jul.). CLOC. Count lines of code [Online]. Available: http://cloc.sourceforge.net/

[60] (2015, Jul.). SuperPI [Online]. Available: http://www.superpi.net/

[61] L. Duflot, O. Levillain, B. Morin, and O. Grumelard, “Getting into the SMRAM: SMM reloaded,” in *Proc. 12th CanSecWest Conf.*, 2009.

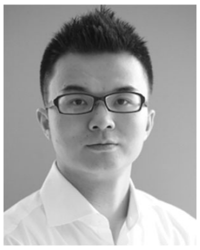
[62] L. Duflot, O. Levillain, B. Morin, and O. Grumelard. (2010). System management mode design and security issues [Online]. Available: http://www.ssi.gouv.fr/IMG/pdf/IT_Defense_2010_final.pdf

[63] R. Wojtczuk and C. Kallenberg. (2014). Attacking UEFI Boot Script. 31st Chaos Communication Congress (31C3) [Online]. Available: http://events.ccc.de/congress/2014/Fahrplan/system/attachments/2566/original/venamis_whitepaper.pdf

[64] J. Butterworth, C. Kallenberg, and X. Kovah, “BIOS Chronomancy: Fixing the core root of trust for measurement,” in *Proc. 20th ACM Conf. Comput. Commun. Security*, 2013, pp. 25–36.

[65] A. M. Azab, P. Ning, and X. Zhang, “SICE: A hardware-level strongly isolated computing environment for x86 multi-core platforms,” in *Proc. 18th ACM Conf. Comput. Commun. Security*, 2011, pp. 375–388.

[66] R. Wojtczuk and J. Rutkowska, “Attacking intel trust execution technologies,” 2009.



Fengwei Zhang received the PhD degree in computer science from the George Mason University. He is an assistant professor at the Department of Computer Science at Wayne State University. His research interests include trustworthy execution, memory introspection, system integrity checking, and transparent malware debugging.

Kevin Leach received the MSc degree in computer science from the George Mason University in 2013. He is currently working toward the PhD degree in computer engineering at the Department of Electrical and Computer Engineering and the Department of Computer Science at the University of Virginia. His research interests include system security and transparent debugging.

Angelos Stavrou received the PhD degree in computer science from the Columbia University in 2007. He is an associate professor at the Department of Computer Science of George Mason University. His research interests include large systems security & survivability, intrusion detection systems, privacy & anonymity, security for MANETs and mobile devices.

Haining Wang received the PhD degree in computer science and engineering from the University of Michigan at Ann Arbor in 2003. He is a professor at the Department of Electrical and Computer Engineering at the University of Delaware. His research interests lie in the area of security, networking system, and distributed computing.