Using Hardware Features for Increased Debugging Transparency

Fengwei Zhang[1], Kevin Leach[2], Angelos Stavrou[1], Haining Wang[3], and Kun Sun[1]

[1]George Mason University, {fzhang4,astavrou,ksun3}@gmu.edu
[2]University of Virginia, kjl2y@virginia.edu
[3]University of Delaware, hnw@udel.edu

***Abstract*—With the rapid proliferation of malware attacks on the Internet, understanding these malicious behaviors plays a critical role in crafting effective defense. Advanced 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 four stepping modes. 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, our prototype of MALT introduces moderate but manageable overheads on both Windows and Linux platforms.**

***Keywords*-malware debugging; transparency; SMM**

I. INTRODUCTION

The proliferation of malware has increased dramatically and caused serious damage for Internet users in the past few years. McAfee reported that the presence of malware has been greatly increasing during the first quarter in 2014, seeing more than 30 million new malware samples [1]. In the last year alone, Kaspersky Lab products detected almost 3 billion malware attacks on user computers, and 1.8 million malicious programs were found in these attacks [2]. Symantec blocked an average of 568,000 web attacks per day in 2013, a 23% increase over the previous year [3]. As such, malware analysis is critical to understanding new infection techniques and maintaining a strong defense.

Traditional malware analysis employs virtualization [4], [5], [6] and emulation [7], [8], [9] 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 [10], [11], [12], [13], [14], [15]. Malware can easily detect the presence of a VM or emulator and alter its behavior to hide itself. Chen et al. [10] executed 6,900 malware samples and found that more than 40% of them reduced malicious behavior under a VM or with a debugger attached. Branco et al. [11] showed that 88% and 81% of 4 million analyzed malware samples had anti-reverse engineering and anti-virtualization techniques, respectively. Furthermore, Garfinkel et al. [16] concluded that virtualization transparency is fundamentally infeasible and impractical. To address this problem, security researchers have proposed analyzing malware on bare metal [17], [18]. This approach makes anti-VM malware expose its malicious behavior, and it does not require any virtualization or emulation technology. However, malware analysis on bare metal runs an analysis program within the Operating System (OS), and ring 0 malware can easily detect its presence. 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 four techniques to provide step-by-step debugging: (1) instruction-level, (2) branch-level, (3) far control transfer level, and (4) near return transfer level. We also design a user-friendly interface for MALT to easily work with several popular debugging clients, such as IDAPro [19] and GDB.

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 [4], [7], [8], [9], which significantly reduces the attack surface of MALT. Moreover, MALT is OS-agnostic and immune to hypervisor attacks (e.g., VM-escape attacks [20], [21]). Compared to existing bare-metal malware analysis [17], [18], SMM has the same privilege level as hardware. Thus, MALT is capable of debugging and analyzing kernel and hypervisor rookits as well [22], [23].

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 four types of step-by-step execution on Windows and Linux platforms. The overhead ranges from 2 to 973 times slowdown on the target system, depending on the user's selected instrumentation method.

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.
- Through testing MALT against popular packers, anti-debugging, anti-virtualization, and anti-emulation techniques, we demonstrate that MALT remains transparent and undetected.

The remainder of this paper is organized as follows. Section II provides background on SMM and BIOS. Section III surveys related work. Section IV discusses our threat model and assumptions. Section V presents the architecture of MALT. Section VI details the implementation of MALT. Section VII analyzes the transparency of MALT. Section VIII shows the performance evaluation of our prototype. Section IX discusses the limitations of MALT. Section X concludes the paper and discusses future directions.

TABLE I
SUMMARY OF SMM FEATURES

SMM Entry	Asserting an SMI
SMM Exit	Running an `RSM` instruction
Memory Access	SMRAM is inaccessible from other CPU modes
Memory Addressing	Physical memory addressing without paging
Interrupts & Exceptions	Disabled upon entering SMM
Privilege	Highest privilege access to all memory & devices

II. BACKGROUND

A. System Management Mode

System Management Mode (SMM) [24] 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 initialized by the Basic Input/Output System (BIOS).

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[1]. The `RSM` instruction forces the CPU to exit from SMM and resume execution in the previous mode. Table I shows a summary of SMM features.

B. 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 [25]. Coreboot performs some hardware initialization and then executes a payload (e.g., UEFI). MALT uses SeaBIOS [25] as the 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.

[1]For this reason, SMM is often referred to as *ring -2*.

TABLE II
COMPARISON WITH OTHER DEBUGGERS

	MALT	**BareBox** [17]	**V2E** [7]	**Anubis** [8]	**Virt-ICE** [9]	**Ether** [4]	**VAMPiRE** [26]	**SPIDER** [5]	**IDAPro** [19]
No VM/emulator	✓	✓					✓		✓
Debug ring0 malware	✓		✓	✓	✓	✓		✓	
Trusted code base	***BIOS***	OS	KVM+QEMU	QEMU	QEMU	Xen	OS	KVM	OS
SLOC of TCB (K)	**1.5**	16,281	13,397	786	786	509	16,281	12,593	16,281

III. RELATED WORKS

A. Malware Debugging and Analysis

VAMPiRE [26] 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 3 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 [4] 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 [27] and Anubis [8] are QEMU-based malware analysis systems. They focus on understanding malware behaviors, instead of achieving better transparency. V2E [7] combines both hardware virtualization and software emulation. HyperDbg [6] 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 [5] uses Extended Page Tables to implement invisible breakpoints 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 [10], [13], [15], [28] 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 [29] has demonstrated that malware running in the guest OS can detect the presence of Ether using CPUID bits, while MALT remains transparent. In terms of transparency, as it relates to the attack surface, MALT has a smaller trusted computing base than hypervisor-based malware analysis systems. Table II shows the trusted computing base of various malware analysis systems. We can see that MALT has a much smaller attack surface than those hypervisor-based systems.

BareBox [17] is a malware analysis framework based on a bare-metal machine without any virtualization or emulation technologies. However, it only targets the analysis of user-mode malware, while MALT is capable of debugging hypervisor rootkits and kernel-mode device drivers. Willems et al. [18] 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 [30] 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.

Virt-ICE [9] 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. However, since it uses a VM, a malware may refuse to unpack itself in the VM. MALT relies on the BIOS integrity and does not employ any virtualization. Thus, we are capable of achieving higher transparency while debugging and analyzing code.

There is a vast array of popular debugging tools. For instance, IDA Pro [19] and OllyDbg [31] are popular debuggers running within the operating system focusing on ring 3 malware. DynamoRIO [32] is a process virtualization system implemented using software code cache techniques. It executes on top of the OS and allows users to build customized dynamic instrumentation tools. Similar to MALT, WinDbg [33] uses a remote machine to connect to the target machine using serial or network communications. However, these options require special booting configuration or software running within the operating system, which is easily detected by malware. Table II summarizes the differences between MALT and other malware debugging and analysis systems. The source lines of code (SLOC) are obtained from [34], and we use the Linux kernel as the OS in Table II.

B. 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 [35], HyperCheck [36], and HyperSentry [37] are integrity monitoring systems based on SMM. SPECTRE [38] 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 [39], [40]. 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 [41] demonstrated the first SMM-based attack to bypass the protection mechanism in OpenBSD. Other SMM-based attacks focus on achieving stealthy rootkits [42], [43]. 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 [44]. However, these attacks require bypassing or unlocking SMRAM protection. MALT locks the SMRAM in the BIOS code. We will discuss bypassing SMRAM protection in Section IX.

IV. Threat Model and Assumptions

A. Threat Model

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.

1) Armored Malware: Armored malware or evasive malware [30] 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 [12]. 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 [10], [14], [15]. For instance, Red Pill [28] 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) [7]. This invalid access causes a General Protection (GP) exception on a bare-metal machine, but QEMU does not.

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 operating system kernel (in 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 [22] and BluePill [23]. 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 [20], [21]). However, for firmware rootkits run in ring -2, MALT cannot detect these kind of rootkits.

B. 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 [45]. We also assume the firmware is trusted, although we can use SMM to check its integrity [46]. After booting, we lock the SMRAM to ensure the SMI handler code is trusted. We discuss attacks against SMM in Section IX. We assume the debugging client and remote machine are trusted. Furthermore, we consider an attacker that can have unlimited computational resources on our machine. We assume the attacker launches a single vulnerable application that can compromise the OS upon completing its first instruction. Lastly, we assume the attacker does not have physical access to the machines. Malicious hardware (e.g., hardware trojans) is also out of scope.

V. System Architecture

Figure 1 shows the architecture of the MALT system. The debugging client is equipped with a simple GDB-like debugger. 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 Figure 1, the debugging client first sends an SMI triggering message to the debugging server; we reroute a serial interrupt to generate an SMI when the message is received. Secondly, once the debugging server enters SMM, the debugging client starts to send debugging commands to the SMI handler on the server. Thirdly, the SMI handler transparently executes the requested commands (e.g., list registers and set breakpoints) and sends a response message back to the client.

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 will detail each component of the MALT system.

A. Debugging Client

The client can ideally implement a variety of popular debugging options. For example, we could use the SMI handler

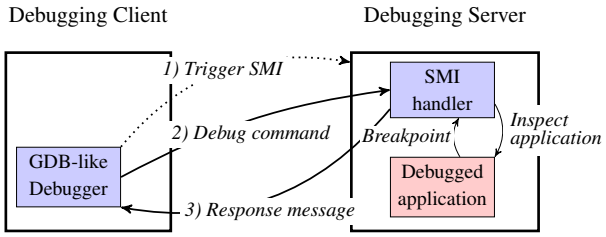

Fig. 1. Architecture of MALT

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. Section X 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 our 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.

B. Debugging Server

The debugging server consists of two parts — the SMI handler and the debugging target application. 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 (SMM). The debugging target executes in Protected Mode as usual. 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 [47]. 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.

C. Communication

In order to implement remote debugging in our system, we define a simple communication protocol used by the client and server hosts. Table III shows the communication protocol commands. These 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.

VI. Design and Implementation

The MALT system is composed of two main parts: (1) the debugging client used by the malware analyst and (2) the debugging server, which contains the SMI handler code and the target debugging application. In this section, we describe how these two parts are implemented and used.

A. 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 Table III. We did not implement features such as symbols; such advanced features pose an engineering challenge that we will address in our future work. The client machine uses serial messages to communicate with the server.

B. 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.

C. Semantic Gap Reconstruction

As with VMI systems [48], SMM-based systems encounter the well-known semantic gap problem. In brief, 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 [47].

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 0x34 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. Figure 2 illustrates this procedure.

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 III
COMMUNICATION PROTOCOL COMMANDS

Message format	Description
R	A single byte, R is sent to request that all registers be read. This includes all the x86 registers. The order in which they are transmitted corresponds with the Windows trap frame. The response is a byte, r, followed by the registers $r_1r_2r_3r_4...r_n$.
mAAAALLLL	The byte m is sent to request a particular memory address for a given length. The address, A, is a 32-bit little-endian virtual address indicating the address to be read. The value L represents the number of bytes to be read.
Wr1r2r3...rn	The byte W is sent to request that the SMI handler write all of the registers. Each value r_i contains the value of a particular register. The response byte, + is sent to indicate that it has finished.
SAAAALLLLV...	The command, S, is sent when the debugger wants to write a particular address. A is the 32-bit, little-endian virtual address to write, L represents the length of the data to be written, and V is the memory to be written, byte-by-byte. The response is a byte, +, indicating that the operation has finished, or a − if it fails.
BAAAA	The B command indicates a new breakpoint at the 32-bit little-endian virtual address A. The response is + if successful, or − if it fails (e.g., trying to break at an already-broken address). If the SMI handler is triggered by a breakpoint (e.g., the program is in breakpoint debugging status), it will send a status packet with the single character, B, to indicate that the program has reached a breakpoint and is ready for further debugging. The SMI handler will wait for commands from the client until the Continue command is received, whereupon it will exit from SMM.
C	The C command continues execution after a breakpoint. The SMI handler will send a packet with single character, +.
X	The X command clears all breakpoints and indicates the start of a new debugging session.
KAAAA	The K command removes the specified breakpoint if it was set previously. The 4-byte value A specifies the virtual address of the requested breakpoint. It responds with a single + byte if the breakpoint is removed successfully. If the breakpoint does not exist, it responds with a single −.
SI, SB, SF, SN	The SI command indicates stepping the system instruction by instruction. The SB command indicates stepping the system by taken branches. The SF command indicates stepping the system by control transfers including far call/jmp/ret. The SN command indicates stepping the system by near return instructions. The SMI handler replies with single character, +.

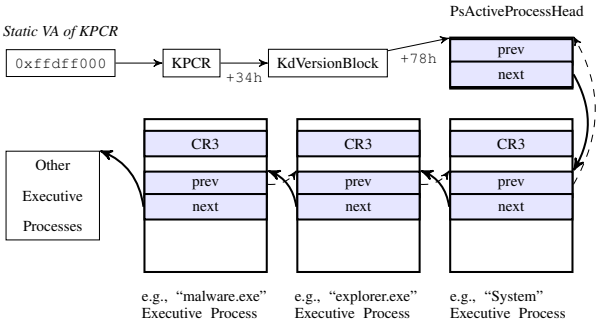

Fig. 2. Finding a Target Application in Windows

table of that process. We use the name field in the EProcess or task_struct 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. Alternatively, we can compare the EIP value with the target application's entry point. This method is simpler but less reliable since multiple applications may have the same entry point. Filling the semantic gap in Linux is a similar procedure, but there are fewer structures and thus fewer steps. Previous works [38], [49] 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 [47]. As with VMI systems, MALT does not consider the attacks that mutate kernel structures.

D. 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 (0x2b on Intel, and 0x52f in our chipset [50]) 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 alter-

native 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) [51]. 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. Lastly, we configure the Local Vector Table Entry (LVTE) in LAPIC to deliver SMIs when an overflow occurs. Similar methods [37], [52] are used to switch from a guest VM to the hypervisor VMX root mode.

E. 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. Stealthy breakpoint insertion is an open problem [26].

In MALT, we emulate the behavior of software breakpoints simply by modifying the target's code to trigger SMIs. An SMI is triggered on our testbed by writing a value to the hardware port, 0x52f. In total, this takes six bytes. We thus save six bytes from the requested breakpoint address and replace them with the SMI triggering code. Thus, when execution reaches this point, the CPU enters SMM. We store the breakpoint in SMRAM, represented as 4 bytes for the address, 6 bytes for the original instruction, and one byte for a validity flag. Thus, each breakpoint occupies 11 bytes in SMRAM. When the application's control reaches the breakpoint, it generates an SMI. In the SMI handler, we write the saved binary code back to the application text and revert the Extended Instruction Pointer (EIP) register so that it will resume execution at that same instruction. Then, we wait in the SMI handler until the client sends a *continue* command. In order to remove an inserted breakpoint, the client can send a remove-breakpoint command and the SMI handler will disable that breakpoint by setting the enable flag to 0. However, this software breakpoint solution still makes changes to the application memory, which is visible to malware. Thus, MALT does not use software breakpoints.

1) Breakpoints in MALT*:* We implement a new hardware breakpoint technique in MALT. It relies on 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. In contrast to the software breakpoint method described above, we do not need to store instructions because we do not change any application memory. 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 [51]. 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.

F. 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 IV 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

TABLE IV
STEPPING METHODS IN MALT

Performance Counter Events	**Description** [51]
Retired instructions	Counts retired instructions, plus exceptions and interrupts (each count as one instruction)
Retired taken branches	Includes all types of architectural control flow changes, including exceptions and interrupts
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

increase as well, so we account for that change accordingly. The system becomes deadlocked if the value is not chosen correctly.

Vogl and Eckert [52] 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 [52] 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 four 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.

VII. TRANSPARENCY ANALYSIS

In terms of transparency, it heavily depends on its subjects. 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 [28] 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 [13]. nEther [29] detects hardware virtualization using CPU design defects. Furthermore, there are many footprints introduced by virtualization such as well-known strings in memory [10], magic I/O ports [17], and invalid instruction behaviors [14]. Moreover, Garfinkel et al. [16] 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 [15]. Table V shows more anti-emulation techniques. Although some of these defects could be fixed, determining perfect emulation is an undecidable problem [4].

SMM: As explained in Section II, SMM is a hardware feature existing in all x86 machines. Regarding its transparency, the Intel manual [24] 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 [19], [26], emulation-based [8], [27], and virtualization-based [4], [5] 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 [4] 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 VII-A and attempt to meet the second requirement in our system. Since MALT runs on bare metal, it immediately meets the third and fourth requirements. Lastly, MALT partially satisfies the fifth requirement by adjusting the local timers in the SMI handler. We further discuss the timing attacks below.

A. 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 (MSR) [24]. 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 [51] 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. Intel recently introduced System Management Range Registers (SMRR) [24] that limits cache references of addresses in SMRAM to code running in SMM. This is the vendor's response to the cache poisoning attack [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 [54], and it takes about 28 seconds to flash 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 this attack. As stated in Ether [4], defending against external timing attacks for malware analysis systems is Turing undecidable. 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.

B. 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 [10], [12], [13], [14], [15] 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 V summarizes popular anti-debugging, anti-virtualization, and anti-emulation techniques, and we have verified that MALT can evade all these detection techniques.

C. 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.

TABLE V
SUMMARY OF ANTI-DEBUGGING, ANTI-VM, AND ANTI-EMULATION TECHNIQUES

Anti-debugging [11], [12]	
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 [10] *checkvm* software [55] 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 [29] CPU errata: AH4 erratum [29]
Other	LDTR register [13] IDTR register (Red Pill [28]) Magic I/O port (0x5658, 'VX') [17] Invalid instruction behavior [14] Using memory deduplication to detect various hypervisors including VMware ESX server, Xen, and Linux KVM [56]
Anti-emulation	
Bochs	Visible debug port [10]
QEMU	`cpuid` returns less specific information [7] Accessing reserved MSR registers raises a General Protection (GP) exception in real hardware; QEMU does not [15] Attempting to execute an instruction longer than 15 bytes raises a GP exception in real hardware; QEMU does not [15] Undocumented `icebp` instruction hangs in QEMU [7], while real hardware raises an exception Unaligned memory references raise exceptions in real hardware; unsupported by QEMU [15] Bit 3 of FPU Control World register is always 1 in real hardware, while QEMU contains a 0 [7]
Other	Using CPU bugs or errata to create CPU fingerprints via public chipset documentation [15]

TABLE VI
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

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 [31], DynamoRIO [57], and a Windows virtual machine, 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 VI 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.

D. Transparency of MALT

Functions and Code Added by MALT: Section VII-B and VII-C 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 VII-A. Note that mitigating all footprints requires stepping instructions granularity. 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.

VIII. EVALUATION

A. Testbed Specification and Code Size

We evaluate MALT on 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 booting. The debugging client is a Dell Inspiron 15R laptop with Ubuntu 12.04 LTS. It uses a 2.4GHz Intel Core i5-2430M CPU and 6 GB DDR3 RAM. We use a USB-to-serial cable to connect two machines.

We use `cloc` [58] to compute the number of lines of source code. Coreboot and its SeaBIOS payload contain 248,421 lines. MALT adds about 1,500 lines of C code in the SMI hander. After compiling the Coreboot code, the size of the image is 1MB, and the SMI hander contains 3,098 bytes. The debugger client contains 494 lines of C code.

B. Debugging with Kernels and Hypervisors

To demonstrate that MALT is capable of debugging kernels and hypervsiors, we intentionally crash the OS kernels and domain 0 of a Xen hypervisor and then use MALT to debug them. For the Linux kernel and domain 0 of the Xen hypervisor, we simply run the command *echo c > /proc/sysrq-trigger*, which performs a system crash by a NULL pointer dereference. To force a Blue Screen of Death (BSOD) in Windows, we create a new value named CrashOnCtrlScroll in the registry key *HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\i8042prt\Parameters* and set it equal to a *REG_DWORD* value of 0x01. Then, the BSOD can be initiated by holding the Ctrl key and pressing the Scroll Lock key twice. After a system crashes, MALT can start a debugging session by sending an SMI triggering message. In our experiments, MALT is able to examine all the CPU registers and the physical memory of the crashed systems.

C. Breakdown of Operations in MalT

In order 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 (TSC) 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

TABLE VII
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		

reconfigure the performance counter registers for the next SMI. Table VII shows a breakdown of the operations in the SMI handler if the last running process is the target malware in the instruction-by-instruction stepping mode. This experiment shows the mean, standard deviation, and 95% 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. Lastly, SMM resume takes 4.58 microseconds. Thus, MALT takes about 12 microseconds to execute an instruction without debugging command communication.

D. Step-by-Step Debugging Overhead

In order to demonstrate the efficiency of our system, we measure the performance overhead of the four stepping methods on both Windows and Linux platforms. We use a popular benchmark program, SuperPI [59] 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. Additionally, we use four popular Linux commands, `ls`, `ps`, `pwd`, and `tar`, to measure the overhead. `ls` is executed with the root directory; `pwd` is executed under the home directory; and `tar` is used to compress a hello-world program with 7 lines of C code. We install `Cygwin` on Windows to execute these commands. First, we run the programs and record their runtimes. Next we enable each of the four stepping methods separately and record the runtimes. SuperPI calculates 16K digits of π, and we use shell scripts to calculate the runtimes of the Linux commands.

Table VIII shows the performance slowdown introduced by the step-by-step debugging. The first column specifies four different stepping methods; the following five columns show the slowdown on Windows, which is calculated by dividing the current running time by the base running time; and the last five columns show the slowdown on Linux. It is evident that far control transfer (e.g., call instruction) stepping only introduces a 2x 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 973x slowdown on Windows for running SuperPI, which demonstrates the worst-case performance degradation in our four debugging methods. This high runtime overhead is due to the 12-microsecond cost of every instruction (as shown in Table VII) in the instruction-stepping mode. 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 three times as fast as Ether [4], [7] in the single-stepping mode.

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. 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.

IX. DISCUSSION AND LIMITATIONS

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 slow. The bootless approach must manually reinitialize the system state, including memory and disks, but takes less time. BareBox [17] used a rebootless approach to restore the memory and disk of the analysis machine; BareCloud [30] used LVM-based copy-on-write to restore a remote storage disk. For the rebootless approach, besides memory and disk restoration, hardware devices also need to 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. MALT simply reboots the analysis machine and reimages the disk and BIOS by copying and reflashing. Additionally, the focus of MALT is to address the debugging transparency problem, while fast restoration (i.e., bootless approach) of a system increases the efficiency of malware analysis. We leave this for future work.

MALT uses SMM as the foundation to implement various debugging functions. Before 2006, computers did not lock their SMRAM in the BIOS [42], and researchers used this flaw to implement SMM-based rootkits [41], [42], [43]. 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 [35] or cache poisoning [53]. The memory reclaiming attack can be 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 [60], but it has been fixed by Intel adding SMRR [24]. Furthermore,

TABLE VIII
STEPPING OVERHEAD ON WINDOWS AND LINUX (UNIT: TIMES OF SLOWDOWN)

Stepping Methods	**Windows**					**Linux**				
	π	ls	ps	pwd	tar	π	ls	ps	pwd	tar
Retired far control transfers	2	2	2	3	2	2	3	2	2	2
Retired near returns	30	21	22	28	29	26	41	28	10	15
Retired taken branches	565	476	527	384	245	192	595	483	134	159
Retired instructions	973	880	897	859	704	349	699	515	201	232

TABLE IX
SUMMARY OF SMM ATTACKS AND SOLUTIONS

SMM Attacks	**Solutions**
Unlocked SMRAM [41], [42], [43]	Set D_LCK bit
SMRAM reclaiming [35]	Lock remapping and TOLUD registers
Cache poisoning [53], [60]	SMRR
Graphics aperture [61]	Lock TOLUD
TSEG location [61]	Lock TSEG base
Call/fetch outside of SMRAM [61], [63]	No call/fetch outside of SMRAM

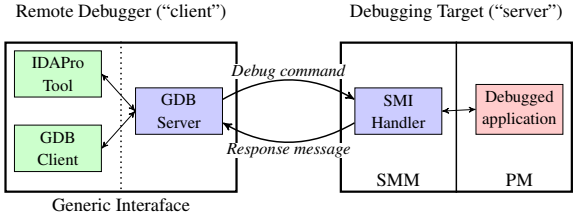

Fig. 3. Using MALT with Multiple Debugging Clients

Duflot et al. [61] listed some design issues of SMM, but they can be fixed by correct configurations in BIOS and careful implementation of the SMI handler. Table IX shows a summary of attacks against SMM and their corresponding solutions. Wojtczuk and Kallenberg [62] 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. The UEFI boot script is a data structure interpreted by UEFI firmware during S3 resume. When the boot script executes, system registers like BIOS_NTL (SPI flash write protection) or TSEG (SMM protection from DMA) are not set so that attackers can force an S3 sleep to take control of SMM. Fortunately, as stated in the paper [62], 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 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 [24]. 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 [37] and SICE [65] are implemented on multi-core processors. In a multi-code 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 [24]. 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 (SMM) 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.

X. CONCLUSIONS AND FUTURE WORK

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 does not require the level of trust associated with hypervisors or operating systems. Thus, 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 could work with multiple debugging clients, such as IDAPro and GDB. MALT introduces moderate but manageable overheads on Windows and Linux, which range from 2 to 973 times slowdown, depending on the stepping method.

We plan to combine MALT with the IDAPro or GDB clients. Our goal is to provide a standard, generic interface for multiple debugging clients to use MALT. First, we will run a `gdbserver` instance along with IDAPro or GDB clients on the remote client so that these debugging clients can use standard protocols to connect to the `gdbserver`. Next, we will modify the `gdbserver` to connect to the SMI handler on the target server by using the protocol defined in Table III. This approach moves the `gdbserver` from the target server to the remote client, and the real GDB stubs will be implemented in the SMI handler instead of the `gdbserver`. We will expand our current protocol to fully support this method, and Figure 3 illustrates this particular usage of MALT.

XI. ACKNOWLEDGEMENTS

We would like to thank our shepherd, Niels Provos, and the anonymous reviewers for their insightful comments that improved the paper. This work is supported by the National Science Foundation Grant No. CNS 1421747 and II-NEW 1205453, Defense Advanced Research Projects Agency Contract FA8650-11-C-7190, and ONR Grant N00014-13-1-0088. Opinions, findings, conclusions and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the US Government or the Navy.

REFERENCES

[1] McAfee, "Threats Report: First Quarter 2014," http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q1-2014-summary.pdf.

[2] Kaspersky Lab, "Kaspersky Security Bulletin 2013," http://media.kaspersky.com/pdf/KSB_2013_EN.pdf.

[3] Symantec, "Internet Security Threat Report, Vol. 19 Main Report," http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_v19_21291018.en-us.pdf, 2014.

[4] 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 (CCS '08)*, 2008.

[5] Z. Deng, X. Zhang, and D. Xu, "SPIDER: Stealthy Binary Program Instrumentation and Debugging Via Hardware Virtualization," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC'13)*, 2013.

[6] A. Fattori, R. Paleari, L. Martignoni, and M. Monga, "Dynamic and Transparent Analysis of Commodity Production Systems," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'10)*, 2010.

[7] L.-K. Yan, M. Jayachandra, M. Zhang, and H. Yin, "V2E: Combining Hardware Virtualization and Software Emulation for Transparent and Extensible Malware Analysis," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE'12)*, 2012. [Online]. Available: http://doi.acm.org/10.1145/2151024.2151053

[8] Anubis, "Analyzing Unknown Binaries," http://anubis.iseclab.org.

[9] N. A. Quynh and K. Suzaki, "Virt-ICE: Next-generation Debugger for Malware Analysis," in *Black Hat USA*, 2010.

[10] 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 *Proceedings of the 38th Annual IEEE International Conference on Dependable Systems and Networks (DSN '08)*, 2008.

[11] R. R. Branco, G. N. Barbosa, and P. D. Neto, "Scientific but Not Academical Overview of Malware Anti-Debugging, Anti-Disassembly and Anti-VM Technologies," in *Black Hat*, 2012.

[12] N. Falliere, "Windows Anti-Debug Reference," http://www.symantec.com/connect/articles/windows-anti-debug-reference, 2010.

[13] D. Quist and V. Val Smith, "Detecting the Presence of Virtual Machines Using the Local Data Table," http://www.offensivecomputing.net.

[14] E. Bachaalany, "Detect If Your Program is Running inside a Virtual Machine," http://www.codeproject.com/Articles/9823/Detect-if-your-program-is-running-inside-a-Virtual.

[15] T. Raffetseder, C. Kruegel, and E. Kirda, "Detecting System Emulators," in *Information Security*. Springer Berlin Heidelberg, 2007.

[16] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin, "Compatibility is not Transparency: VMM Detection Myths and Realities," in *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems (HotOS'07)*, 2007.

[17] D. Kirat, G. Vigna, and C. Kruegel, "BareBox: Efficient Malware Analysis on Bare-metal," in *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC'11)*, 2011.

[18] 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 *Proceedings of the Annual Computer Security Applications Conference (ACSAC'12)*, 2012.

[19] IDA Pro, www.hex-rays.com/products/ida/.

[20] K. Kortchinsky, "CLOUDBURST: A VMware Guest to Host Escape Story," in *Black Hat USA*, 2009.

[21] R. Wojtczuk, J. Rutkowska, and A. Tereshkin, "Xen 0wning Trilogy," in *Black Hat USA*, 2008.

[22] S. T. King and P. M. Chen, "SubVirt: Implementing Malware with Virtual Machines," in *Proceedings of the 27th IEEE Symposium on Security and Privacy (S&P'06)*, May 2006.

[23] J. Rutkowska, "Blue Pill," http://theinvisiblethings.blogspot.com/2006/06/introducing-blue-pill.html, 2006.

[24] 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

[25] Coreboot, "Open-Source BIOS," http://www.coreboot.org/.

[26] A. Vasudevan and R. Yerraballi, "Stealth Breakpoints," in *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC'05)*, 2005.

[27] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A New Approach to Computer Security via Binary Analysis," in *Proceedings of the 4th International Conference on Information Systems Security (ICISS'08)*, 2008.

[28] J. Rutkowska, "Red Pill," http://www.ouah.org/Red_Pill.html.

[29] G. Pek, B. Bencsath, and L. Buttyan, "nEther: In-guest Detection of Out-of-the-guest Malware Analyzers," in *Proceedings of the 4th European Workshop on System Security (EuroSec'11)*, 2011.

[30] D. Kirat, G. Vigna, and C. Kruegel, "BareCloud: Bare-metal Analysis-based Evasive Malware Detection," in *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[31] OllyDbg, www.ollydbg.de.

[32] D. Bruening, Q. Zhao, and S. Amarasinghe, "Transparent Dynamic Instrumentation," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE'12)*, 2012.

[33] Windbg, www.windbg.org.

[34] Ohloh, "Black Duck Software, Inc," http://www.ohloh.net, access time: 10/16/2014.

[35] J. Rutkowska and R. Wojtczuk, "Preventing and Detecting Xen Hypervisor Subversions," http://www.invisiblethingslab.com/resources/bh08/part2-full.pdf, 2008.

[36] F. Zhang, J. Wang, K. Sun, and A. Stavrou, "HyperCheck: A Hardware-assisted Integrity Monitor," in *IEEE Transactions on Dependable and Secure Computing (TDSC'14)*, 2014.

[37] 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 *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, 2010.

[38] F. Zhang, K. Leach, K. Sun, and A. Stavrou, "SPECTRE: A Dependable Introspection Framework via System Management Mode," in *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13)*, 2013.

[39] J. Wang, F. Zhang, K. Sun, and A. Stavrou, "Firmware-assisted Memory Acquisition and Analysis Tools for Digital Forensic," in *Proceedings of the 6th International Workshop on Systematic Approaches to Digital Forensic Engineering (SADFE '11)*, 2011.

[40] A. Reina, A. Fattori, F. Pagani, L. Cavallaro, and D. Bruschi, "When Hardware Meets Software: A Bulletproof Solution to Forensic Memory Acquisition," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC'12)*, 2012.

[41] L. Duflot, D. Etiemble, and O. Grumelard, "Using CPU System Management Mode to Circumvent Operating System Security Functions," in *Proceedings of the 7th CanSecWest Conference (CanSecWest'04)*, 2004.

[42] S. Embleton, S. Sparks, and C. Zou, "SMM rootkits: A New Breed of OS Independent Malware," in *Proceedings of the 4th International Conference on Security and Privacy in Communication Networks (SecureComm'08)*, 2008.

[43] BSDaemon, coideloko, and D0nAnd0n, "System Management Mode Hack: Using SMM for 'Other Purposes'," *Phrack Magazine*, 2008.

[44] "NSA's ANT Division Catalog of Exploits for Nearly Every Major Software/Hardware/Firmware," http://Leaksource.wordpress.com.

[45] Trusted Computing Group, "TCG PC Client Specific Implementation Specification for Conventional BIOS, Specification Version 1.21," http://www.trustedcomputinggroup.org, February 2012.

[46] F. Zhang, H. Wang, K. Leach, and A. Stavrou, "A Framework to Secure Peripherals at Runtime," in *Proceedings of The 19th European Symposium on Research in Computer Security (ESORICS'14).*, 2014.

[47] B. Jain, M. B. Baig, D. Zhang, D. E. Porter, and R. Sion, "SoK: Introspections on Trust and the Semantic Gap," in *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P'14)*, 2014.

[48] T. Garfinkel and M. Rosenblum, "A Virtual Machine Introspection Based Architecture for Intrusion Detection," in *Proceedings of the 10th Annual Network and Distributed Systems Security Symposium (NDSS'03)*, 2003.

[49] 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)*, 2007.

[50] VIA Technologies, Inc., "VT8237R South Bridge, Revision 2.06," December 2005.

[51] Advanced Micro Devices, Inc., "BIOS and Kernel Developer's Guide for AMD Athlon 64 and AMD Opteron Processors," http://support.amd.com/TechDocs/26094.PDF. [Online]. Available: http://support.amd.com/us/ProcessorTechDocs/26094.PDF

[52] S. Vogl and C. Eckert, "Using Hardware Performance Events for Instruction-Level Monitoring on the x86 Architecture," in *Proceedings of the 2012 European Workshop on System Security (EuroSec'12)*, 2012.

[53] R. Wojtczuk and J. Rutkowska, "Attacking SMM Memory via Intel CPU Cache Poisoning," 2009. [Online]. Available: http://invisiblethingslab.com/resources/misc09/smm_cache_fun.pdf

[54] Flashrom, "Firmware Flash Utility," http://www.flashrom.org/.

[55] checkvm: Scoopy doo, http://www.trapkit.de/research/vmm/scoopydoo/scoopy_doo.htm.

[56] J. Xiao, Z. Xu, H. Huang, and H. Wang, "Security Implications of Memory Deduplication in a Virtualized Environment," in *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13)*, 2013.

[57] DynamoRIO, "Dynamic Instrumentation Tool Platform," http://dynamorio.org/.

[58] CLOC, "Count lines of code," http://cloc.sourceforge.net/.

[59] SuperPI, http://www.superpi.net/.

[60] L. Duflot, O. Levillain, B. Morin, and O. Grumelard, "Getting into the SMRAM: SMM Reloaded," in *Proceedings of the 12th CanSecWest Conference (CanSecWest'09)*, 2009.

[61] ——, "System Management Mode Design and Security Issues," http://www.ssi.gouv.fr/IMG/pdf/IT_Defense_2010_final.pdf.

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

[63] R. Wojtczuk and A. Tereshkin, "Attacking Intel BIOS," https://www.blackhat.com/presentations/bh-usa-09/WOJTCZUK/BHUSA09-Wojtczuk-AtkIntelBios-SLIDES.pdf.

[64] J. Butterworth, C. Kallenberg, and X. Kovah, "BIOS Chronomancy: Fixing the Core Root of Trust for Measurement," in *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS'13)*, 2013.

[65] A. M. Azab, P. Ning, and X. Zhang, "SICE: A Hardware-level Strongly Isolated Computing Environment for x86 Multi-core Platforms," in *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*, 2011.

[66] R. Wojtczuk and J. Rutkowska, "Attacking Intel Trust Execution Technologies," 2009.