SPECTRE: A Dependable Introspection Framework via System Management Mode

Fengwei Zhang, Kevin Leach, Kun Sun and Angelos Stavrou

Center for Secure Information Systems
George Mason University
Fairfax, VA 22030
{fzhang4, kleach2, ksun3, astavrou}@gmu.edu

***Abstract*—Virtual Machine Introspection (VMI) systems have been widely adopted for malware detection and analysis. VMI systems use hypervisor technology for system introspection and to expose malicious activity. However, recent malware can detect the presence of virtualization or corrupt the hypervisor state thus avoiding detection.**

We introduce SPECTRE, a hardware-assisted dependability framework that leverages System Management Mode (SMM) to inspect the state of a system. Contrary to VMI, our trusted code base is limited to BIOS and the SMM implementations. SPECTRE is capable of transparently and quickly examining all layers of running system code including a hypervisor, the OS, and user level applications. We demonstrate several use cases of SPECTRE including heap spray, heap overflow, and rootkit detection using real-world attacks on Windows and Linux platforms. In our experiments, full inspection with SPECTRE is 100 times faster than similar VMI systems because there is no performance overhead due to virtualization.

Keywords*—*SMM, introspection, memory attacks.

I. INTRODUCTION

The complexity and ever-increasing sophistication of malware coupled with our everyday dependence on computer systems has resulted in an ongoing arms race in the cyber-security field. Today, malicious code can both infect and hide its activities effectively, even while exfiltrating data and damaging the victim host. To make matters worse, Internet-borne malware is focusing more than ever on taking over desktop applications and installing host rootkits. For example, variants of ZeroAccess [1], a rootkit for both 32-bit and 64-bit versions of Windows, is quickly becoming one of the most widespread malware threats.

Traditionally, malware detection is provided by installing anti-malware tools (e.g., anti-virus) within the operating system. However, all defensive techniques that run as processes in the operating system are inherently vulnerable to malicious code executing at the same level. Therefore, when a rootkit compromises the OS, most if not all of the common protection suites become ineffective, misleading the user that the system is protecting while the malware operates freely in the background.

To address this, security researchers suggested Virtual Machine Introspection (VMI) [2] for malware detection. VMI executes all programs inside a guest Virtual Machine (VM), translating their semantic state information to malware detection tools that run outside the VM (i.e., on the host). The goal is to isolate and protect the malware detection software from the potentially vulnerable guest so that stealthy malware cannot interfere or corrupt the protection mechanisms. Although a step to the right direction, VMI systems have practical limitations.

First, VMI systems depend on the integrity of the hypervisor, which has a sizable Trusted Computing Base (TCB). For instance, the latest Xen 4.2 contains approximately 208K lines of code. Although this size is dwarfed by the code size of a typical operating system, the attack surface posed by the hypervisor remains significant. The National Vulnerability Database [3] shows that there are 100 vulnerabilities in Xen and 90 vulnerabilities in VMWare ESX. In addition, VM escape attacks [4], [5] and hypervisor rootkits [6] are widely deployed.

Secondly, armored malware can detect the presence of a VM or debugger and alter its own execution [7], [8], [9]. Indeed, malware running inside of the VM can read a virtual device name or simply read the IDT or LDT registers to detect the presence of a VM [10]. In such cases, attaining behavioral transparency from the perspective of the malware is urgent and difficult to achieve.

Lastly, and most importantly, traditional VMI techniques incur a high overhead on system performance, making them unpopular among end-users. Some of the more theoretical solutions are deemed incur such a high latency that they are deemed unfit for use in current computing systems. For instance, existing VMI methods often take on the order of seconds to pause a VM guest to scan its memory. We could potentially improve the performance through the use of heuristics or other approximations, but this leads to false positives and false negatives, adversely affecting the end solutions.

In light of these problems, we have developed SPECTRE, a novel framework capable of attaining *fast, OS-level transparency with a small TCB* while examining a live system. We do not rely on a large hypervisor, instead employing a small (fewer than 1,000 lines) code base. Thus, software running at the OS level (e.g., rootkits) cannot be made aware of our system while maintaining high performance. SPECTRE leverages System Management Mode (SMM) on x86-based architectures to identify malware at different levels (hypervisor, OS, process) without any dependence on the underlying code. Since SMM code is part of the BIOS, SPECTRE only needs to

trust the BIOS and any volatile system firmware, thus heavily reducing the size of the TCB. Therefore, assuming that we can trust the hardware and BIOS, the framework remains sound. We demonstrate potential applications of our approach by detecting a variety of memory-based attacks without depending on the integrity of underlying software components while the system is running. Using our system, we expose the presence of heap spray attacks, heap overflow attacks, and rootkits. SPECTRE enables OS-transparent introspection of *native* system memory. Thus, armored malware using anti-virtualization will be unable to detect our system.

As an enabling platform for other malware detection techniques, SPECTRE offers a fast solution to the semantic gap problem. It can quickly reconstruct data structures used by the kernel about each process running on a system. In this paper, we use our introspection system to detect a variety of stealthy malware. In particular, we demonstrate successful exposure of heap spray attacks, heap overflow attacks, and various rootkits, all on the native system without using a VMM or other abstraction technology. We also explain how this modular framework can be augmented for use in future work. As such, we consider our system to be not only a useful detection mechanism and introspection tool, but also a vehicle for enabling research related to advanced malware.

In our experiments using working prototypes for Microsoft Windows and Linux systems, we were able to find and reconstruct process-related structures in less than 8ms in Windows and less than 5ms in Linux. Our semantic reconstruction enabled us to transparently defend against a wide range of malware threats including heap spray, heap overflow, and rootkit activity. To that end, we detected samples of heap spray attacks in less than 35ms. In Windows, we were able to detect heap overflows in 32ms. We were also able to detect a rootkit in Windows in 8ms, while the same process took 5ms in Linux. Lastly, we demonstrate SPECTRE is capable of maintaining a reasonably low performance overhead by varying the sampling interval.

Contributions In summary, we make the following contributions:

- We introduce a hardware-assisted framework that can examine code across all layers of running system code including a hypervisor, the operating system, and individual applications.
- SPECTRE is OS-agnostic and fully transparent to higher level software including malware.
- We have implemented a prototype of our framework in both Linux and Windows, and demonstrated that our system can detect various memory attacks including heap spray, heap overflow and rootkits.
- Compared to existing VMI approaches we can achieve faster execution for both detection and regular application execution while relying on a much smaller TCB.
- SPECTRE is transparent to all code running on the targeted host enabling us to perform transparent analysis avoiding detection by the malicious code.

II. BACKGROUND

A. System Management Mode

System Management Mode (SMM) is an execution mode similar to Real and Protected modes available on the x86 architectures. It provides a transparent mechanism for implementing platform-specific system control functions such as power management. It is implemented by the BIOS.

SMM is triggered by asserting the system management interrupt (SMI) pin on the motherboard hardware. This pin can be asserted using both software and hardware mechanisms. After assertion, the CPU saves its state in a specific region of memory called system management RAM (SMRAM). Next, it atomically executes the system management interrupt (SMI) handler. SMRAM is inaccessible from other CPU execution modes, and can therefore act as trusted storage. The SMI handler has access to physical address space (i.e., paging is disabled) and can run any instruction. We expand upon SMM and its usage in Sections IV and V.

B. BIOS and Coreboot

The Basic Input-Output System (BIOS) is an integral part of all computers. It initializes hardware and loads the operating system. BIOS code is stored on non-volatile ROM on the motherboard. Coreboot [11] is an open-source project aimed at replacing the BIOS in current computers. It performs a small hardware initialization, then executes its 'payload.' Coreboot switches to protected mode at an early stage and is chiefly written in the C language. SPECTRE uses a custom SMI handler in Coreboot. This made implementation much easier since Coreboot already handles hardware initialization. Furthermore, this allows SPECTRE to be far more portable to other systems, since Coreboot abstracts away heterogeneity of specific hardware configurations.

III. THREAT MODEL AND ASSUMPTIONS

A. Memory-based Attacks

Most malware will alter memory at some point causing the system to enter a state not intended in the original design. We call them *memory-based* attacks. For example, a typical heap spray attack will place a large number of NOP instructions in dynamic memory, a heap overflow attack will change heap application data or metadata in memory, and rootkits will alter kernel code or data. Since SPECTRE is capable of examing all layers of running system code and data (e.g., hypervisor, OS, user-level applications) in the memory, it can sucessfully detect those memory-based attacks including heap spray, heap overflow, and rootkits when accommodating corresponding memory checking modules.

B. Assumptions

SPECTRE uses SMM to detect malware in the operating system. The attack is assumed to have unlimited computing resources and can exploit zero-day vulnerabilities of desktop applications. We have a similar threat model to VMI systems as in [12], [13], [14], but since we do not rely on the operating system or hypervisor to accomplish the inspection task, we do not need to trust the hypervisor or the OS. We assume SMM is locked and will remain intact after boot, and the attacker cannot

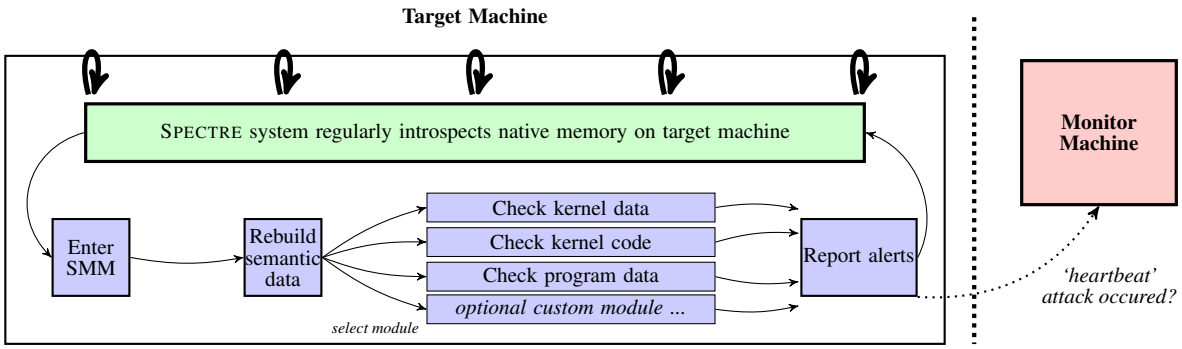


Fig. 1. Operation of SPECTRE. We can detect a variety of memory-based attacks from within the SMI handler.

change the SMI handler or flash the BIOS and reboot. Cache poisoning techniques which change the SMI handler [15] are out of the scope of this paper. We assume that the target machine is equipped with trusted boot hardware, such as a BIOS with Core Root of Trust for Measurement (CRTM) and a Trusted Platform Module (TPM) [16] to ensure the integrity of the SMI handler upon booting the system. We assume that the attacker does not have physical access to the machine. We also assume that the hardware can be trusted to function normally; malicious hardware (e.g., hardware trojans) is out of scope.

IV. System Architecture

Fig. 1 illustrates the operation of the SPECTRE system, which consists of two machines. The *target machine* is the machine to be protected, and the *monitor machine* is responsible for receiving status messages from the target machine and triggering alerts. The whole inspection process consists of four major stages. First, the target machine enters SMM by triggering a system management interrupt (SMI) regularly and reliably. Second, after entering SMM, the target machine rebuilds accurate semantic information about the operating system in a trusted environment without relying on the (potentially altered) operating system. Third, the target machine executes monitoring modules that evaluate the integrity of the kernel or user-space processes. Finally, a "heartbeat" message is sent securely to the monitor machine. When a suspicious behavior is detected, an alert is transmitted as part of the heartbeat message.

SMM gives us several useful properties. First, it has unrestricted access to the whole physical memory in the system, so it is difficult for stealthy, malicious code to hide itself. Second, the SMI handler is loaded only once when the computer is powered up, and locked thereafter. Thus, even if malicious code can rewrite the BIOS or SMI handler, it won't be able to execute that code until rebooting. However, we can use TPM to prevent this attack by checking the integrity of the BIOS and SMM code before booting up. Third, the SMI handler can quickly inspect memory because it executes atomically and benefits tremendously from locality optimizations. Lastly, SMM provides a region of secure memory (SMRAM) in which we store data each time the SMI handler runs. This secure memory allows us to inspect system memory without having to trust the underlying operating system for storing relevant data. This protection is ensured transparently by the memory management unit (MMU), which redirects accesses to SMRAM addresses to a portion of video memory.

A. Periodic Triggering of System Management Mode

We periodically assert a system management interrupt (SMI) on the target machine so that it can enter SMM. There are two ways to trigger a SMI: software-based and hardware-based. Software can cause a SMI via I/O access to a particular port specified by the chipset. Software-based mechanisms are convenient and easy to implement, but they are not transparent to the operating system. Thereforew, if the OS becomes compromised, malicious code can interfere with software and prevent it from accessing those special ports.

Alternatively, many hardware devices are also capable of triggering a SMI, including PCI devices, keyboards, and hardware timers. Our system utilizes a hardware timer built into the chipset, which is capable of generating a SMI at a regular and configurable interval. We set the timer configuration parameters in the BIOS before the OS loads, so we can trust the timer after booting. Nonetheless, advanced malware may be able to change these configuration settings after compromising the OS. This would effectively result in a denial of service. However, a monitor machine can trivially detect denials of service—it expects "heartbeat" messages sent at regular intervals. If these heartbeats cease, then the monitor machine can detect a denial of service attack. Additionally, we can prevent masquerading attacks by using a key exchange in the BIOS before the OS loads. We describe the detailed implementation in Section V.

B. Rebuilding Semantic Information

Since SMM has unrestricted access to all physical memory and registers, once entering SMM, the target machine can introspect all its physical memory. However, since only physical address space is visible in SMM, we must reconstruct the semantics of various operating system structures in order to evaluate data and code integrity of the kernel and user-space programs.

When the SMI handler is first triggered, the CPU context that is saved contains virtual addresses relevant to the running thread. Moreover, in both Windows and Linux, many kernel structures reference virtual addresses. Thus, we should first be able to translate virtual addresses to physical addresses, so that we can access important data from within the SMI handler. Fortunately, this process is identical in both Windows and Linux.

Both operating systems reserve a large section of virtual address space for kernel mode operations. Addresses above

a constant, PAGE_OFFSET, are considered kernel space. PAGE_OFFSET is 0xC0000000 for Linux, and 0x80000000 for Windows. In either case, finding the physical address in the kernel space simply consists of subtracting PAGE_OFFSET from the virtual address.

For user space virtual addresses (VA), both operating systems use a two-level paging scheme on our test bed. There is a page directory containing pointers to particular page tables, which in turn point to specific pages. Each 32-bit virtual address is split into three regions: a 10-bit page directory offset, a 10-bit page table offset, and a 12-bit page offset. The CR3 register points to the base of the page directory. From there, the first 10 bits of the virtual address are used to find an entry in the directory which points to the base of a page table. The next 10 bits of the virtual address are used as an offset into the page table, yielding a pointer to the page where the required data is stored. Physical Address Extension (PAE) essentially adds a fourth level of translation which could easily be integrated into our system.

C. Memory Checking Modules

Our system is designed to easily accommodate various existing defensive technologies. We demonstrate this capability with several *modules* that detect an array of attacks including Heap Spray, Heap Overflow, and Rootkits. Other checking modules can be extended into the SMI handler of our system. More details can be found in Section V.

1) Heap Spray Attacks: SPECTRE's heap spray detection module regularly scans the heaps of vulnerable processes in memory. When a heap spray attack occurs, it will fill the heap with a NOP sled. When detecting a large region of NOP bytes, we conclude that a heap spray attack has occurred.

2) Heap Overflow Attacks: SPECTRE can detect heap overflow attacks by evaluating the integrity of heap-related structures in the operating system. Typically, heap overflow attacks will alter entries in the *free list* maintained by the operating system [17]. This structure helps the OS track which blocks of the heap have been freed by the program for reallocation. A heap overflow attack will overflow the boundary of a heap buffer and rewrite data contained in an adjacent free block. This behavior will cause inconsistencies in the free list for which we can easily scan.

3) Rootkits: Once installed, rootkits pose a serious threat to a system's health. Nonetheless, in order to execute any code, they must alter the system memory in some detectable way, such as corrupting the list of processes. Ultimately, we detect rootkits by evaluating the integrity of kernel structures.

D. Reporting Alerts

Once detecting an attack, the SMI handler alerts the monitor machine over a serial or Ethernet cable. We must ensure that communicating with the monitor machine is secure. There are two requirements to establish a trusted connection between the target machine and the monitor machine: a shared secret key between the target and the monitor machines, and a trusted network interface on the target machine. The target machine can establish a shared secret key with the monitor machine in the BIOS before booting the OS. Since we trust the BIOS at startup, we can store the key in the trusted SMRAM. This key is then rendered inaccessible from other execution modes; only our SMI handler has access to it. This allows us to ensure that an attack cannot masquerade as our system to the monitor machine.

Since we cannot rely on the target machine's untrusted operating system to relay the alert message to the monitor machine, our approach involves writing driver code within the SMI handler for our network card. We use two separate network interfaces in our testbed—one for normal network usage, and one exclusively for use by our SMI handler. This approach makes our system more transparent to the operating system. With no driver installed in the OS, software is unaware of its presence. Our system can remain undetected while operating, naturally at the expense of a PCI slot. However, while a compromised OS can scan the PCI device and write a new driver to operate the network card, it still cannot fake the network packet without the shared secret key. Thus, reporting alerts of SPECTRE is secure.

Moreover, our system can detect denial of service attacks that may occur if our system becomes compromised. Since the monitor machine will expect receiving "heartbeat" updates from the target machine at regular intervals, it is trivial to detect aberrations in packet delivery time.

V. DESIGN AND IMPLEMENTATION

SPECTRE supports both Windows and Linux OS environments. In our testbed, the target machine has a ASUS-M2V_MX SE motherboard with AMD K8 northbridge and VIA VT8237R southbridge, 2.2GHz AMD Sempron LE-1250 CPU, and 2GB Kingston DDR2 RAM. We use the integrated network card for normal network traffic and an Intel e1000 Gigabit network card for SMM packet transmission. Additionally, the monitor machine consists of a simple Linux machine. It runs an instance of minicom for communication via the serial port and a simple socket program to receive network packets from the target machine. Its specifications are inconsequential to the performance of the system. We use an open source BIOS, Coreboot with a SeaBIOS payload. For a Linux environment, we use CentOS 5.5 with kernel 2.6.24 and Debian with kernel 2.6.32. For a Windows environment, we use Windows XP SP3. Each environment is 32 bit; however, our system is also capable of running in a 64-bit environment with slight changes in the paging system.

We now describe the design and implementation of each of the four steps mentioned in Section IV: triggering SMM at regular intervals, reconstructing semantic data, running a detection module, and communicating with the monitor machine.

A. Periodically Triggering SMM

We use a hardware timer, General Purpose 0 (GP0), to periodically assert a system management interrupt (SMI) [18]. The timer is configured via control registers in the southbridge, which we set in Coreboot before the OS loads. This timer is configured with a starting value and unit of time. When the specified unit of time elapses, the timer value decrements by 1. Upon reaching 0, the timer will assert a SMI and then reset its value, restarting the process again. For example, when we

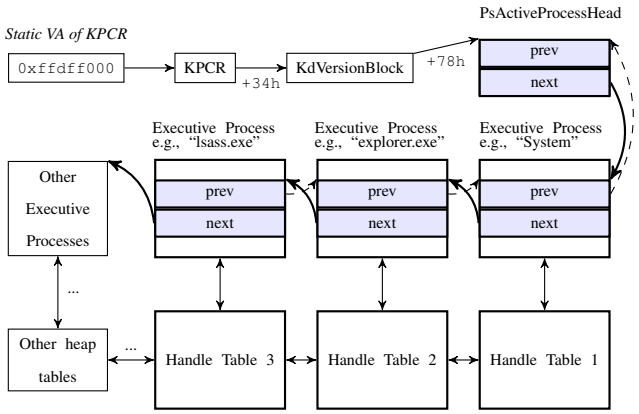


Fig. 2. Finding the list of processes in Windows.

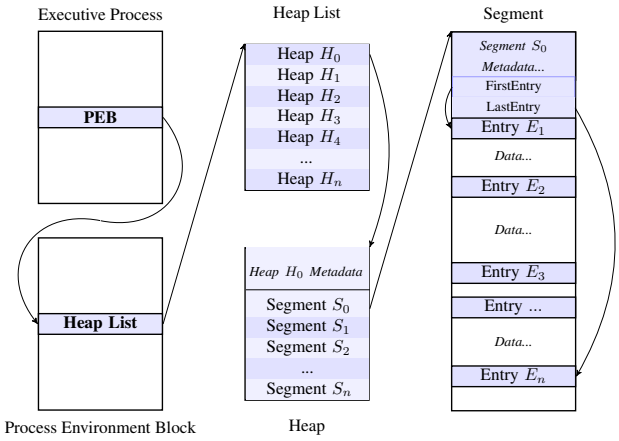


Fig. 3. Finding heap data in Windows. Each arrow represents a translation from virtual to physical space.

assign the timer a value of 5 and a time unit of 1 second, it will trigger a SMI every 5 seconds.

Though a software-based SMI triggering mechanism would be easier to use, it is not usable for two reasons. First, software-based triggering would require extending our trust base to the operating system—malware that compromises the OS is able to stop the triggering software. Secondly, since it is software, the exact timing is left to the mercy of the OS scheduler. If the software trigger does not get scheduled due to high multiprogramming in the OS, then we may unintentionally suffer a denial of service. Thus, we choose the hardware-based approach that is much more reliable and transparent.

B. Rebuilding Semantic Information

SPECTRE performs inspection operations within SMM, which is agnostic to the particular operating system. However, in order to introspect the integrity of the OS, we have to fill the semantic gap on the data structures, which are different for Windows and Linux environments. We elaborate on this below.

1) Bridging the Semantic Gap in Microsoft Windows: Microsoft Windows maintains a complex hierarchy of data structures responsible for processes and threads. In particular, each CPU is associated with a Kernel Processor Control Region (KPCR) [19]. This data is always present at a static virtual address, 0xffdff000, in memory. Thus, if we can translate that virtual address to a physical address, we can easily access it from within the SMI handler.

Fortunately, the CR3 register stores the physical address of the page directory of the currently executing process. This allows us to find the physical address corresponding to any given virtual address used by that process, including the KPCR. While each process has a unique CR3 value, SMM saves the CR3 register when switching from protected mode. Therefore, we can simply read the CR3 value from SMRAM to find the KPCR regardless of what the OS was doing before the SMI occurred.

At offset 0x34 of the KPCR, there is a pointer to another structure, the `KdVersionBlock`, which contains certain global variables pertinent to the current version of the kernel. Within the `KdVersionBlock`, the address of `PsActiveProcessHead` is stored at offset 0x78. `PsActiveProcesssHead` is a pointer to the start of a circularly- and doubly-linked list of pointers to executive process structures, which we then use to find the heap of each process. Additionally, the executive process structure provides forward and backward links to other executive process structures. Fig. 2 provides a visual representation of the structures we must traverse to find `PsActiveProcessHead` and the executive process structures to which it links.

Unfortunately, some rootkits are capable of altering this linked list to hide themselves through a technique called Direct Kernel Object Manipulation (DKOM) process hiding. This means we must consider alternative ways of enumerating processes. We consider a method used by a program called KProcCheck [20].

First, we can use the method above to find the first executive process running on the system (called the `PsInitialSystemProcess`). This executive process is located at a fixed address, so we only need to find it once when first starting the system. Even if a rootkit removes this process from the linked list, we can still retrieve it from within the SMI handler.

Next, within the executive process structure, there is a `HANDLE_TABLE` structure containing information about that process's files, devices, ports, and similar handle objects. This structure contains a `HandleTableList` consisting of backward and forward links to handle tables of other processes. This means we can enumerate each handle table for every process running on the system, regardless of whether or not a given process has been hidden. Additionally, the `HANDLE_TABLE` structure also contains a pointer to the executive process to which it belongs. Thus, even if a rootkit uses DKOM hiding, we can still find the executive process it tries to hide using this method.

Using the methods described above, we can enumerate all of the processes running on the system. Each executive process structure exists in kernel space of a particular process. It contains the name of the binary on the filesystem (e.g., "firefox.exe"). This allows us to find and analyze a specific process, regardless of which process is running when SMM is triggered. While rootkits would be able to change the name of the process, we would be able to detect it via simple integrity

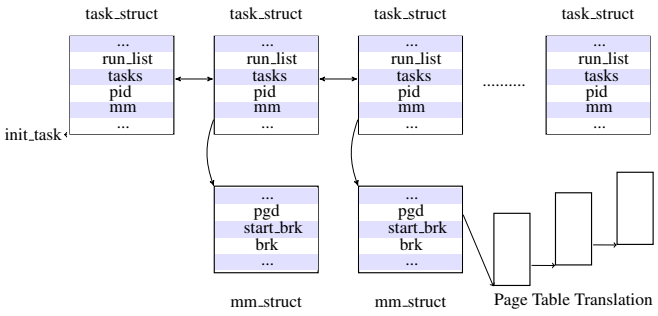


Fig. 4. Finding the list of tasks in Linux.

checking. We can simply store the name of the image in SMRAM and check if that same executive process changed names during the next run.

Each executive process contains a pointer to a Process Environment Block (PEB), which is a user space structure that stores the locations of heap structures belonging to that process. Each process has at least one default heap, and can optionally create additional private heaps as needed. Pointers to each heap are stored in the process's PEB. Each heap structure contains additional pointers to a maximum of 64 heap segments, each of which stores a sequence of heap entries. The entries contain 8 bytes of metadata followed by the actual heap data. The entries in a segment are stored contiguously in virtual memory. Fig. 3 illustrates the hierarchy of data structures we must traverse to enumerate entries in the heap.

2) Filling the Semantic Gap in Linux: Linux offers a much simpler process management scheme. There is a circularly- and doubly-linked list of `task_struct` structures, each of which contains information about processes running in the system. We can enumerate all of the processes by finding a single `task_struct` and walking through the list.

We leverage the kernel-exported symbol information to find a starting `task_struct` and then enumerate all of the processes. Firstly, we find the virtual address of the `init_task` pointer from the `System.map` file in the `/boot` directory. The `System.map` file is produced once when the kernel is compiled; it stores all of the symbol information about the kernel. `init_task` is a static address that points to the `task_struct` of a specific process, `swapper`. Within this `task_struct`, we can find forward and backward links that form a circularly- and doubly-linked list of tasks at offset 0x178 in our kernel. Additionally, offset 0x29b contains the name of the process, which helps identify specific processes. Fig. 4 illustrates these steps to find the list of tasks in Linux. Similarly, we use the `modules` symbol in `System.map` to find the list of kernel modules.

Since the `task_struct` list is used by the scheduler, Linux rootkits cannot hide by altering this list—otherwise, they might impact their own execution. Instead, they often hide by altering the `/proc` directory.

Once we have the pointer to a task, all information related to the process address space is included in an object called the memory descriptor, `mm`. The `mm` field stores the pointer to a memory structure, called `mm_struct`, for the process. The `mm_struct` structure contains `start_brk` and `brk` fields, which correspond to the starting and ending addresses of the heap.

In contrast to Windows, the Linux environment simply allocates heap space one page at a time (via the `sbrk` system call). Typically, applications use heap allocators built into libraries like glibc, and thus malware typically exploits vulnerabilities in a particular heap allocator. For example, glibc uses a similar free list structure as in Windows—16 byte total metadata in free entries with forward and backward links to other free entries of the same size. Thus, the glibc allocator has free list vulnerabilities similar to those in Windows.

C. Running a Detection Module

Once we glean relevant semantic information from the operating system, we can start executing a module for system inspection. Our system is flexible to easily accommodating various existing defensive technologies. We demonstrate this capability with several modules that can detect various memory-based attacks, including heap spray attacks, heap overflow attacks, and rootkits. Note that we acknowledge the simplicity of these detection algorithms that should not be considered as major contributions of this paper; our goal is to show the flexibility of our system framework to accommodate various checking modules. Other checking modules can be extended into the SMI handler of our system.

1) Detecting Heap Spray Attacks: Once we have access to heap data, detecting a heap spray is the same for both Windows and Linux environments. We scan the heap for the presence of a potential NOP sled. Unfortunately, the x86 NOP instruction, 0x90, is not the only technique used to achieve NOP-like behavior. Other common instructions include `or al, 0x0c` and `xor eax, eax`. In fact, many repeated sequences of bytes exhibit the behavior of a NOP sled, provided they do not affect the registers required for the shellcode to execute. Therefore, we heuristically check for the presence of a NOP sled by searching for contiguous, repeated sequences of bytes in the heap of a process.

Essentially, we wrote a regular expression engine in the SMI handler which recognizes the following pattern: `[^(0x00|0xFF)]{n,}`. This pattern will recognize a sequence of at least n or more repeated bytes other than 0x00 or 0xFF. Naturally, changing the value of n will affect the false positive rate. The results of our experiments are detailed in section VI.

2) Detecting Heap Overflow Attacks: In Windows, an application can have multiple heaps, and each heap has a free list array with 128 elements called the FreeList. We can find this array at offset 0x178 from the heap base. Each FreeList is a list of free chunks chained by a doubly linked-list. Each free chunk has 16 bytes meta data including sizes of and pointers to the previous and current free chunks. In Linux, heap management is provided by a library (e.g. glibc), but the free blocks are chained by doubly-linked lists and uses the same 16 byte header structure. The attacks exploiting the FreeList depend on the specific heap implementation, but the malicious code must change pointers to hijack execution. Our system transverses all entries in each heap's free list to see if there is any broken points.

We did not implement a heap overflow detection module for Linux because heap free blocks are maintained by glibc library. This adds another layer of the semantic gap problem for reconstructing heap structures. We considered it as a future work.

3) Detecting Rootkits: Detecting rootkits depends upon monitoring the integrity of 1) kernel code, and 2) kernel data. To check the integrity of kernel code, we simply compute a hash of the static kernel code within the SMI handler. Alternatively, we send the static kernel code to a remote server for integrity attestation. Remote checking may be favorable in environments where consumption of network bandwidth is less expensive than local hash computation. Since the SMI handler essentially pauses the native system, we want to avoid overly long computation in the SMI handler to avoid incurring too much overhead.

The more challenging aspect is maintaining integrity of dynamic data structures in the kernel. Previous research has proposed many defensive techniques against rootkits [12], [21], [13]. In order to demonstrate this capability in our system, we wrote a simple rootkit detection module of listing all running processes (`pslist`) and kernel modules (`lsmod`) for both Microsoft Windows and Linux platforms. We leverage the security caveats of SMM to bring accurate semantic information from the operating system to a trusted 'external' viewpoint. We can discover rootkits by comparing these external views with the internal views of the operating system process states. In Section VI, we test our system using some rootkits available in the wild.

D. Communication with the Monitor Server

The last stage of our system requires communicating with an external server. We accomplish this task by writing driver code for our particular network card in the SMI handler. It consists of manually configuring registers on the device and interacting with the PCI bus.

In brief, we implemented a simple MAC-layer protocol for communicating with the external server. It sends a 214 byte packet in the SMI handler consisting of a 14 byte header and a fixed 200 byte payload. The payload is encrypted with a simple XOR with a key we store in SMRAM, which is first retrieved before the OS loads. The payload contains a sequencing number which simply increments by 1 each time the SMI handler runs. The rest of the payload provides enough spaces for detection modules to convey specific information to the monitor machine.

VI. Experimental Results

A. Code Size

First, we considered the size of the code required for our system to run. In total, there are 470 lines of new C code in the SMI handler, including all three memory checking modules. Each module consisted of less than 100 lines of C code, and the total network transmission code was 110 lines. After compiling the Coreboot code, the binary size of our SMI handler was only 780 bytes, which reduce the trusted computing base of our system.

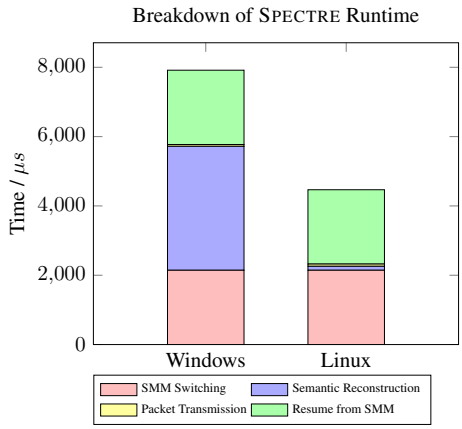


Fig. 5. Breakdown of SMI handler runtime.

B. Breakdown of SMI Handler Runtime

Next, it is important to quantify how much time is required to execute each step of our system. For this experiment, we have broken SPECTRE into the following logical operations: 1) Switching to SMM; 2) Reconstructing OS semantics; 3) Running a detection module; 4) Reporting status via NIC; 5) Switching from SMM to resume the OS. All of the above except for step 3 should take constant amounts of time. The running time of a detection module will depend upon the type of attack and the complexity of the detection technique. For example, the time taken to traverse a linked list of processes will depend upon how many processes are running (i.e., the length of the list) when the module begins executing. However, the rest of the steps execute a fixed set of instructions, and thus we expect them to have somewhat constant running times. In this section, we wanted to understand the 'fixed cost' associated with using our system. In other words, *how much time does* SPECTRE *need to bring useful semantic information to the developer?* Thus, we considered each of the times associated with steps 1, 2, 4, and 5. Step 3 (running detection modules) is discussed later.

We measured the time taken by each step by measuring the TSC register, which stores how many CPU cycles have elapsed since powering on. We disabled technologies in the BIOS that affected CPU clock speed so that a difference in the TSC register represented a constant unit of time, computed with the equation,

$$T = (R_1 - R_0)(\frac{1}{C}),$$

where T is measured time, R_t is the value of the TSC register at time t, and C is the clock speed on the CPU. We recorded the TSC register at several points during our system's execution, such as the beginning and end of the SMI handler.

Fig. 5 shows the observed times taken for each step in each operating system. We can see from the graph that switching to and resuming from SMM take significant time. This is attributed to the power management operations that SMM must perform before our SMI handler can execute. Similarly, the time to resume from SMM is explained by several factors. Upon resuming from SMM, the hardware must also reconfigure itself to allow subsequent SMIs to occur, which requires

TABLE I. HEAP SPRAY ATTACK DETECTION TIME (N=25)

	Detection Time	STD
Firefox	31.168 ms	0.272 ms
Internet Explorer	27.917 ms	0.154 ms
Adobe Acrobat Reader	25.839 ms	0.302 ms
Adobe Flash	29.455 ms	0.603 ms

many I/O operations thus leading to a considerable running time.

Note the significant difference in the time taken for reconstructing the semantics of each operating system. Reconstructing Windows kernel semantics is much longer than in Linux (by two orders of magnitude). This is mainly due to the page table translation steps required in Windows since so much of the data about processes is stored in userspace. In Linux, however, most of the required data is stored in kernel space, and therefore finding the physical addresses reduces to simple subtraction of the PAGE_OFFSET constant.

C. Heap Spray Detection Module

We implemented a heap spray detection module as described previously. We tested Firefox, Adobe Acrobat Reader, and the Adobe Flash plugin in both Windows and Linux, but since MSIE is not available for Linux, we only tested it in Windows.

We chose four heap spray attacks available as Metasploit modules. Using Metasploit eased the experimentation because it allowed rapid deployment of each attack. Each attack has a corresponding Common Vulnerabilities and Exposures entry. The attacks we used are:

1) Firefox 3.5 CVE-2009-2478
2) Internet Explorer 6, 7, 8 CVE-2010-3971
3) Adobe Acrobat 9, 10.1 CVE-2011-2462
4) Adobe Flash Player $<$ 10.2 CVE-2011-6069

These attacks all exploit a vulnerability in an application that causes it to start executing code in the heap. They are all written in scripting languages. The first three attacks use JavaScript, and the last uses ActionScript. They cause the host program to start executing the malicious script, which causes it to allocate large amounts of memory. Then, the attack hijacks control through another means (use-after-free, stack overflow, etc.) to start executing the sprayed memory. We ran this experiment in Windows and not Linux because the dynamic memory system in Windows is much more complex, and thus provides a 'worst-case' performance figure. Table I shows the average results for 25 trials of each type of heap spray attack. The results shows that SPECTRE can detect these attacks in less than 32 ms.

D. Heap Overflow Detection Module

We tested our system against CVE-2012-0276, a real heap overflow attack affecting an image viewer in Windows called XnView. The vulnerability exists in XnView versions 1.98 and earlier. In XnView, insufficient validation while decompressing certain TIFF files enables a heap-based buffer overflow. The malicious image overflows a heap entry, then it rewrites metadata of nearby free chunks in the heap. Then, it simply waits for these blocks to be reused. When the operating system unlinks one of these free blocks from the FreeList, execution jumps to the shellcode.

We detected this attack by checking the integrity of the FreeList, and it takes 32 ms to detect this attack including 24 ms spent in the detection module and the fixed 8 ms associated with entering and exiting SMM.

E. Rootkit Detection Module

We used real, publicly available rootkits to test out system on both Windows and Linux platforms. On Windows platforms, we devised an effective defense mechanism against the Fu rootkit [22]. Fu Rootkits allow the intruder to hide information from user-mode applications and even from kernel modules. Fu hides information by directly manipulating data structures in the kernel. In particular, it removes an entry from the `PsActiveProcessHeader` list. However, we are able been able to find hidden such processes by finding and traversing the HANDLE_TABLE list.

We successfully detected the Fu rootkit using this method. On the target machine, Fu hides the ssh.exe process. We detected the hidden process by enumerating the handle tables in the SMI handler. This technique took only 8ms.

On the Linux platform, we tested a newly available rootkit, KBeast (Kernel Beast), on kernel 2.6.32. KBeast is an advanced armored Linux rootkit that hides its loadable kernel module, hides files and directories, hides processes, hides sockets and connections, performs keystroke logging, and has anti-kill functionality [23]. It is currently undetectable by the latest rootkit detectors including *chkrootkit* [24] and *rkhunter* [25]. KBeast leverages the `sys_write` system call to fake output of system commands like `ps`, `pstree`, `top`, and `lsof` to hide itself. Again, since SMM has an external view of the system states, our system reconstructs the semantics of data structures from physical memory to detect malicious behavior like process hiding.

We were able to detect KBeast in about 5ms using our system. Using the `ps` within the OS missed a network daemon process for malicious remote access. However, SPECTRE successfully discovered the hidden process by traversing the process list in the system.

F. System Overhead

The last and most important experiment tested how much overhead our system introduces to the target machine. We used freely available benchmarking software for both Microsoft Windows and Linux environments. This helped us account for the impact on overall system performance caused by our system's periodic operation. For this experiment, we ran the benchmarking software without our system in place. Next, we ran the same benchmark with SPECTRE enabled at several different time intervals ranging from 0.0625 to 5.0 seconds using the General Purpose 0 (GP0) hardware timer on the southbridge to periodically trigger a SMI. We then calculated the overhead as a ratio between the scores with and without the system in place.

In this experiment, the heap spray detection module targeted the heap of the Adobe Acrobat Reader application in both Windows and Linux. The heap overflow detection module targeted the heap of the XnView process in Windows. We did not consider a Linux-based heap overflow detection

module because it requires reconstructing another layer of semantic information from the particular heap allocator used by a process. Lastly, the rootkit detection module listed all of the running processes in memory to find hidden processes.

1) Windows Evaluation: In Windows, we used PassMark PerformanceTest to measure benchmark our test system. We specifically ran the CPU, disk, and memory tests in PassMark to see the implications on raw performance. Fig. 6(a) shows the results of this experiment. These results indicate the relatively low overhead introduced at all sampling intervals. From Fig. 6(a), we can see that the heap spray and heap overflow modules have slightly larger overhead than the rootkit detection module. This is because the heap-based modules must scan heap data, which in itself takes roughly 30ms. Rootkit detection, on the other hand, simply scans the list of running tasks in memory. This task takes only 8ms in the SMI handler.

2) Linux Evaluation: We used a similar methodology to test the overhead in Linux. We used the UnixBench suite to test performance while the system ran. This was less geared toward CPU and memory performance, instead focusing on specific Unix-like operations, like system call and shell piping performance. The results are presented Fig. 6(b). In general, SPECTRE introduces low overhead in Linux. Even at the lowest sampling interval of $\frac{1}{16}s$ (62.5ms), it causes only 20% overhead in the heap spray detection module, and only 5% overhead in the rootkit detection module.

G. Comparison with VMI Systems

SPECTRE provides a new framework for transparent system introspection and stealthy malware detection. Compared to well-known virtual machine introspection based architectures [2], the BIOS in SPECTRE serves a role similar to the hypervisor in VMI systems. Theoretically, SPECTRE can achieve the same level of protection as VMI does if 1) we are able to implement and execute the same detection algorithms in SMRAM, and 2) we are able to reconstruct all of the necessary kernel- and user-space data structures that serve as the input to the detection algorithms. In this paper, we showed several ways to include different detection modules and recover the necessary semantic data in SPECTRE.

SPECTRE improves upon VMI systems in three ways. First, SPECTRE is a hardware-assisted introspection tool which relies only on the BIOS—it does not need to trust the large-size hypervisor. Thus, SPECTRE has a much smaller TCB. Second, SPECTRE can achieve better transparency than VMI systems. Nowadays, armored malware [9], [26], [27], [10] can easily detect the presence of a VM, but SPECTRE can remains transparency while monitoring these malware. Third, SPECTRE achieves better performance because it does not need to deal with nested page table translation, and SMM switching is faster than VM switching. Table II shows the runtime comparison between SPECTRE and Virtuoso [13]. The program `pslist` shows all of the running process information in the OS, and the program `lsmod` shows all of the loaded kernel modules. The results show that SPECTRE can run these tools 100 times faster than those in Virtuoso. Recently, hardware virtualization extensions (e.g., Intel VT, AMD-V) have been adopted to VMI systems to speed up the introspection process.

TABLE II. RUNTIME COMPARISON OF INTROSPECTION PROGRAMS BETWEEN SPECTRE AND VIRTUOSO

		SPECTRE (ms)	Virtuoso (ms)
Windows	`pslist`	6.6	450.2
	`lsmod`	7.6	698.1
Linux	`pslist`	4.3	6494.1
	`lsmod`	4.4	2437.0

VMI systems can operate based on trap conditions, allowing asynchronous, event-based tools. The current SPECTRE prototype can only execute periodically, but it is a straightforward engineering challenge to implement similar functionality by using performance counters to trigger SMIs. We can assert SMIs when certain conditions are met in the CPU performance counters. For instance, when the instruction cache miss counter overflows, we can assert a SMI.

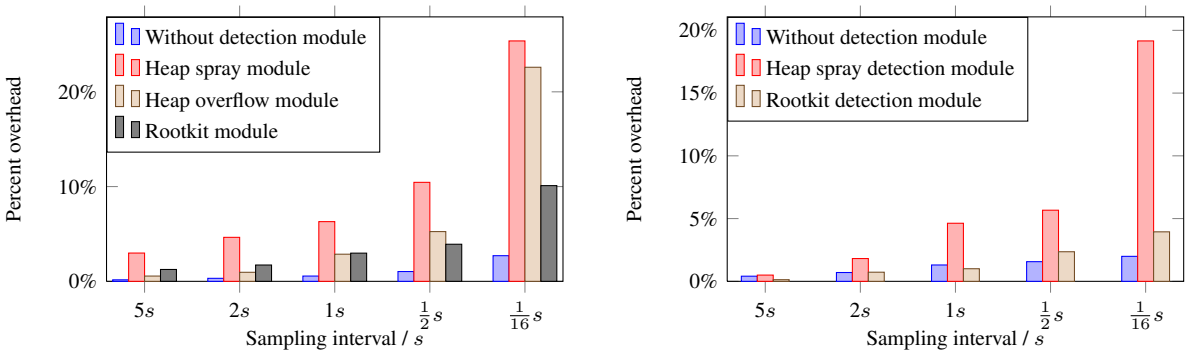
VII. SPECTRE LIMITATIONS AND DISCUSSION

The total potential SMRAM size is large enough to include a number of specific malware detection modules and algorithms. The default size of SMM memory area is 64 KB between 0x30000 and 0x3FFFF, with another 4MB memory region called *TSeg* optionally available as SMRAM [28]. In our testbed, the size of our SMI handler with three detection modules is only 780 bytes. However, the SMRAM is not large enough to accommodate the general anti-virus tools which rely upon large signature databases. We can solve this problem by running complex malware detection tools on the monitor machine, which can request the target machine to send the raw memory data through the network [29], [30]. It will significantly reduce the porting complexity and space used by the SMI handler. However, it will increase the system overhead on the target machine. In this scenario, the SMM is responsible for sending the whole memory (e.g., 4GB in a 32 bit system) to the target machine, the operating system on the target machine is suspended during the packet transmission. It will cause significant delays, ultimately suspending the OS and interrupting the user.

Leveraging existing malware detection algorithms or building customized detection tools requires a considerable human effort due to the semantic gap problem. It is not guaranteed to work correctly if, for example, the OS is significantly changed or updated. For future work, we are considering automatic reconstruction of operating system semantics combined with [13], [14].

In addition, SPECTRE only focuses on attacks in memory, and it does not check the content stored on other I/O devices such as hard disks. Thus, we cannot inspect data that has been swapped to disk. Inspecting the filesystem would require porting a disk driver into the SMI handler, and further filling the semantic gap introduced by the particular hard disk vendor and chosen filesystem.

Since SPECTRE relies on periodically polling system memory, it is vulnerable to the evasion attack [31], where attacker cleans up the trace before introspection begins. To address this problem, we could enter SMM every instruction by using performance counters in the CPU, thus guaranteeing that every state is introspected. We could also use a random scheduler to trigger SMI instead of fixed time interval. In this case, the attacker would not know the time interval used by our system, increasing the accuracy and precision required of the attack.



(a) Overhead introduced in Microsoft Windows

(b) Overhead introduced in Linux

Fig. 6. Overhead introduced in SPECTRE

Vigilare [32] is a system bus traffic monitor on system-on-a-chip (Soc) for checking kernel integrity. It can defend the evasion attack in polling systems by using event-driven monitoring mechanism. However, Vigilare requires extra hardware support (i.e., to duplicate system bus traffic to the verifier) in desktops and servers, while SPECTRE leverages an existing hardware feature (SMM) for malware introspection.

Because SPECTRE relies a on dedicated network card to report a heartbeat message to the monitor machine, malicious NIC firmware may attempt to manipulate the "heartbeat" message. To defend against such attacks, SPECTRE can store a hash value of the NIC firmware in SMRAM and check its integrity before packet transmission in the SMI handler.

Although we implement SPECTRE on a single core processor, SMM is able to deal with multi-core processors as well. Since each core has its own MSR registers, each core can define its own SMRAM memory range which contains the SMI handler. When an SMI is generated, it switches all of the cores on the platform to SMM, each executing its own SMI handler. In order to achieve isolation and transparency, we could let one core execute SPECTRE code, and other cores simply wait until inspection finishes. Another more efficient way is to let one core stay in SMM for introspection while the other cores resume execution in the Protected Mode. However, this approach must handle inter-core communication carefully. SICE [33] has demonstrated this method on AMD processors.

Intel recently introduced SMI Transfer Monitoring (STM) [34] that virtualizes SMM code. The idea is to provide a secured VMM launch. Unfortunately, the use of an STM involves disabling SMIs, thus potentially preventing our system from executing. However, we can modify the STM itself, which executes in SMM, to provide the benefits of SPECTRE without affecting the added security of STM.

SPECTRE is intended to be an extensible framework capable of enabling advanced detection and analysis techniques in a secure environment. We evaluated the system through several use cases such as heap spray attacks. Attack detection techniques such as NOZZLE [35] can be ported to our system to increase their coverage. For example, NOZZLE currently is limited to detecting heap spray attacks within the browser process in userspace. However, we can easily adopt the detection algorithm within SPECTRE to cover attacks in a larger scope. In brief, our system framework facilitates broadening the coverage of analysis and detection.

SPECTRE currently counters three memory-based attacks: heap spray, heap overflow and rootkit attacks. Other types of memory-based attacks (e.g. ROP) could also be detected by SPECTRE after accommodating corresponding efficient detection algorithms as optional detection modules in our system.

VIII. RELATED WORK

A. *Memory-based Attacks and Detection*

Memory-based attack detection is an active research area. In recent years, due to the extensive usage of web browsers on various web applications, more and more heap-based memory corruption attacks have surfaced [36], [37], [17], since attackers can easily allocate malicious objects using scripting languages embedded in a web page. In 2009, heap spraying exploits have been identified in the Adobe Reader using JavaScript embedded in malicious PDF documents [38].

Researchers have proposed a number of effective defensive mechanisms [39], [40] against heap-based memory attacks. DieHarder [41] analyzed several memory allocators and showed they were vulnerable to attack, and also presented a new memory allocator against heap-based attacks. NOZZLE [35], a runtime heap spray detector, examined individual objects in the heap, interpreted them as code, and performed static code analysis to detect malicious intent. Instead of detecting the attacks at the operating system level, [42] can detect drive-by-download attacks by emulation to automatically identify malicious JavaScript code. In this paper, SPECTRE uses some simple yet effective detection algorithms to detect samples of heap spray and heap overflow attacks, which shows the capability and usefulness of our system. Moreover, SPECTRE is designed with the intent to conveniently accommodate more advanced attack detection techniques as new detection modules.

A number of mechanisms and systems have been built to enforce kernel integrity and detect potential rootkits. SecVisor [43] is a tiny hypervisor that leverages new hardware extensions to enforce life-time kernel integrity. However, the deployment of SecVisor requires modification of the kernel. Instead, SPECTRE does not need to change any code in the operating system, although it does require changing the BIOS. Flicker [21] and Trustvisor [44] employ Dynamic Root of Trust Measurement (DRTM) to provide a trust environment for

running security code. One particular usage is to run a rootkit detector for OS integrity checking. SPECTRE can achieve a similar goal by using only SMM.

B. Bridging the Semantic Gap in VMI Systems

The semantic gap problem has fueled a large amount of research [12], [13], [14]. Recently, virtualization has been employed in many environments. Security researchers have embraced virtual machine monitors (VMMs) as a new mechanism to guarantee deep isolation of untrusted software components from the system. "out-of-the-box" defense mechanisms can resist tampering at the cost of a native, semantic view of the host that is enjoyed by the "in-the-box" approach. SPECTRE must solve the same semantic gap problem since it has no context information when the system enters SMM.

VMWatcher [12] is a stealthy malware detection system that uses semantic view reconstruction. Essentially, it pauses a VM guest and scans the memory of that guest and then reconstructs semantic information of data structures. Both Virtuoso [13] and VMST [14] are techniques that can automatically bridge the semantic gap in VM guests. Compared to these methods, SPECTRE must bridge the semantic gap manually and execute different detection modules within the SMI handler. However, our code can run natively without the need of a hypervisor. In other words, our code base is much smaller than VMI systems. Additionally, SPECTRE runs much faster than existing approaches. VMWatcher took on the order of seconds to pause a VM and scan it; Virtuoso took 6 seconds for `pslist` command; and VMST took 60 milliseconds to dump all the pids in the kernel. However, SPECTRE takes only 5 milliseconds to list all the processes in Linux. Moreover, we consider automatically filling semantic gap in SMM as a future work.

C. SMM-based Defensive Systems

SMM has been used as basic building block for several defensive mechanisms. HyperGuard [45] suggests using SMM to monitor hypervisor integrity by taking snapshots of a VM guest and checking it in SMM. HyperCheck [29] had similar goals, but outsourced the snapshot to an external server for OS/hypervisor integrity checking, since it can reduce the computation overhead on the protected machine. HyperSentry [30] used an out-of-band channel, specifically the Intelligent Platform Management Interface, to trigger SMM to check the integrity of base code operating on critical data. All the existing SMM-based defensive solutions focus on enforcing OS or hypervisor integrity checking. Our SPECTRE system provides a new SMM-based introspection framework for memory-based malware detection in both OS level and application level. Besides checking the integrity of the static kernel code, our system can also detect the malware hidden in the dynamic kernel code/data and user-space program code/data. After filling the semantic gap in both Linux and Windows operating systems, SPECTRE can accommodate various detection modules to satisfy specific protection requirements.

Several attacks based on SMM have been proposed too. In 2004, Loic Dulfot [46] developed the first SMM-based attack to bypass protection mechanisms in OpenBSD. Other SMM-based attacks focus on achieving stealthy and efficient rootkits [47], [48]. Most reported vulnerabilites on SMM have been fixed by the manufactures. In this paper, we assume that SMM can be trusted.

IX. CONCLUSIONS

In this paper, we presented SPECTRE, a dependable introspection framework for detecting memory-based stealthy malware by leveraging System Management Mode. It introspects a live operating system without relying on any underlying software, and provides a fast, transparent, secure framework for malware detection with a small TCB. We demonstrate several use cases for SPECTRE including heap spray, heap overflow, and rootkit detection using real-world attacks. Through experiments on both Windows and Linux platforms, we are able to reconstruct process semantic information in under 8ms and under 5ms, respectively; the introspection tools with SPECTRE performed 100 times faster than similar tools in VMI systems.

X. ACKNOWLEDGEMENTS

The authors would like to thank all of the anonymous reviewers for their valuable comments and suggestions. This work is supported by the United States Air Force Research Laboratory (AFRL) through Contract FA8650-10-C-7024. 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 Air Force.

REFERENCES

[1] J. Wyke, "Major shift in strategy for ZeroAccess rootkit malware, as it shifts to user-mode," June 2012. [Online]. Available: http://nakedsecurity.sophos.com/2012/06/06/zeroaccess-rootkit-usermode/

[2] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proceedings Network and Distributed Systems Security Symposium*, 2003.

[3] National Institute of Standards, NIST, "National vulnerability database, http://nvd.nist.gov."

[4] K. Kortchinsky, "CLOUDBURST: AVMwareGuesttoHostEscapeStory," in *BlackHatUSA*, 2009.

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

[6] S. King and P. Chen, "Subvirt: implementing malware with virtual machines," in *Security and Privacy, 2006 IEEE Symposium on*, may 2006, pp. 14 pp. –327.

[7] M.-K. Sun, M.-J. Lin, M. Chang, C.-S. Laih, and H.-T. Lin, "Malware virtualization-resistant behavior detection," in *Proceedings of IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS'11)*, 2011.

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

[9] X. Chen, J. Andersen, Z. 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.

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

[11] "Coreboot." [Online]. Available: http://www.coreboot.org/

[12] 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*, 2007.

[13] T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *Proceedings of the 32nd IEEE Symposium on Security and Privacy*, 2011.

[14] Y. Fu and Z. Lin, "Space Traveling across VM: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, 2012.

[15] 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

[16] Trusted Computing Group, "Trusted Platform Module main specification. version 1.2, revision 103, 2007." [Online]. Available: http://www.trustedcomputinggroup.org/resources/tpm_main_specification

[17] S. Designer, "JPEG COM marker processing vulnerability," July 2000. [Online]. Available: http://www.openwall.com/articles/JPEG-COM-Marker-Vulnerability

[18] VIA, "Vt8237r southbridge." [Online]. Available: http://www.via.com.tw/

[19] E. Barbosa, "Finding some non-exported kernel variables in Windows XP," http://www.reverse-engineering.info/SystemInformation/GetVarXP.pdf.

[20] T. Kong. (2004, May) KProcCheck by SIGˆ2: Win2K kernel hidden process/module checker. [Online]. Available: http://www.security.org.sg/code/kproccheck.html

[21] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for TCB minimization," in *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2008.

[22] "Fu rootkit," 2006. [Online]. Available: http://www.f-secure.com/v-descs/fu.shtml

[23] "KBeast rootkit," 2012. [Online]. Available: http://packetstormsecurity.org/files/108286/KBeast-Kernel-Beast-Linux-Rootkit-2012.html

[24] Chkrootkit, "chkrootkit rootkit detector." [Online]. Available: http://www.chkrootkit.org/

[25] Rkhunter, "rkhunter rootkit detector." [Online]. Available: http://www.rootkit.nl/projects/rootkit-hunter.html/

[26] E. Bachaalany, "Detect if your program is running inside a virtual machines." [Online]. Available: http://www.codeproject.com/Articles/9823/Detect-if-your-program-is-running-inside-a-Virtual

[27] D. Quist and V. Val Smith, "Detecting the presence of virtual machines using the local data table," http://www.offensivecomputing.net/.

[28] 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

[29] J. Wang, A. Stavrou, and A. Ghosh, "HyperCheck: A hardware-assisted integrity monitor," in *Proceedings of 13th International Symposium On Recent Advances In Intrusion Detection*, 2010.

[30] 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*, 2010.

[31] J. Wang, K. Sun, and A. Stavrou, "A dependability analysis of hardware-assisted polling integrity checking systems," in *Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'12)*, 2012.

[32] H. Moon, H. Lee, J. Lee, L. Kim, P. Y., and K. B., "Vigilare: Toward Snoop-based Kernel Integrity Monitor," in *Proceedings of the 19th ACM conference on Computer and communications security (CCS'12)*, 2012.

[33] 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*, 2011.

[34] Intel, "Intel® 64 and IA-32 Architectures Software Developers Manual." [Online]. Available: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

[35] P. Ratanaworabhan, B. Livshits, and B. Zorn, "Nozzle: A defense against heap-spraying code injection attacks," in *Proceedings of the 18th USENIX Security Symposium*, 2009.

[36] Y. Ding, T. Wei, T. Wang, Z. Liang, and W. Zou, "Heap Taichi: exploiting memory allocation granularity in heap-spraying attacks," in *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC '10)*, 2010.

[37] A. Sotirov, "Heap Feng Shui in JavaScript," in *In Black Hat Europe*, 2007.

[38] Multi-State Information Sharing and Analysis Center., "Vulnerability in Adobe Reader and Adobe Acrobat could allow remote code execution." [Online]. Available: http://www.msisac.org/advisories/2009/2009-008.cfm

[39] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda, "Defending browsers against drive-by-downloads: Mitigating heap-spraying code injection attacks," in *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'09)*, 2009.

[40] J. Vanegue, "Zero-sized heap allocations vulnerability analysis,," in *In Proceedings of the 2nd Conference on USENIX Workshop on Offensive Technologies (WOOT'10)*, 2010.

[41] G. Novark and B. E., "Dieharder: securing the heap," in *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, 2010.

[42] M. Cova, C. Kruegel, and G. Vigna, "Detection and analysis of drive-by-download attacks and malicious JavaScript code," in *Proceedings of the19th International World Wide Web Conference (WWW '10)*, 2010.

[43] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," in *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*, 2007.

[44] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "TrustVisor: Efficient TCB reduction and attestation," in *Proceedings of the 31st IEEE Symposium on Security and Privacy*, 2010.

[45] J. Rutkowska and R. Wojtczuk, "Preventing and detecting Xen hypervisor subversions," *Blackhat Briefings USA*, 2008.

[46] 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*, 2004.

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

[48] 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 Netowrks*, 2008.