SHARK: Architectural Support for Autonomic Protection Against Stealth by Rootkit Exploits

Vikas R. Vasisht
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332
vvasisht3@gatech.edu

Hsien-Hsin S. Lee
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332
leehs@gatech.edu

Abstract

Rootkits have become a growing concern in cyber-security. Typically, they exploit kernel vulnerabilities to gain root privileges of a system and conceal malware's activities from users and system administrators without any authorization. Once infected, these malware applications will operate completely in stealth, leaving no trace for administrators and anti-malware tools. Current anti-rootkit solutions try to either strengthen the kernel by removing known vulnerabilities or develop software tools at the OS or Virtual Machine Monitor levels to monitor the integrity of the kernel. Seeing the failure of these software techniques, we propose, in this paper, an autonomic architecture called *SHARK*, or *Secure Hardware support Against RootKit* by employing hardware support to provide system-level security without trusting the software stack, including the OS kernel. SHARK enhances the relationship between the OS and the hardware architecture, making the entire system more security-aware in defending rootkits.

SHARK proposes new architectural support to provide a secure association between each software context and the underlying hardware. It helps system administrators to obtain feedback directly from the hardware to reveal all running processes, even when the OS kernel is compromised. We emulated the functionality of SHARK by using x86 Bochs and modifying the Linux kernel version 2.6.16.33 based on our proposed architectural extension. Several real rootkits were installed to compromise the kernel and conceal malware processes on our emulated environment. SHARK is shown to be highly effective in identifying a variety of rootkits employing different software schemes. In addition, the performance analysis based on our Simics simulations shows a negligible overhead, making the SHARK architecture highly practical.

1. INTRODUCTION

The security of a computing system highly depends on the vulnerability of its underlying operating system. As the complexity of the modern OS increases, its vulnerability to attacks also multiplies. To warrant a highly secure computing system, the kernel security becomes crucial. Currently, thrusts towards providing kernel-level security are mainly in the software, either through the redesign of an OS or adding security patches to reduce the number of known vulnerabilities. Yet designing a monolithic OS with zero vulnerability is unrealistic due to its growing complexity and sheer size, often in many million lines of codes. On the other hand, many research activities are devoted to the design of intrusion detection systems that aim at detecting malware by checking the integrity of various critical software components periodically. Although these techniques are partially successful in strengthening a system's security, defending against malware in the software stack is neither a proactive nor an effective solution for desirable security. This is because, the vulnerable software stack is a common battle ground for both malware and intrusion detection systems — both run with the same privilege, trying to counteract each other. In fact, it is always a losing battle for software intrusion detection systems as it is not difficult for malware to circumvent the defense mechanism based on software, which results in a never ending loop in providing more software solutions for new threats attacking different vulnerabilities. To address these issues more effectively, it is necessary to enhance the relationship between the OS and the hardware architecture, making hardware more security-aware, in particular when interacting with the OS.

Rootkits are gaining more attention these days, as they are detrimental, tenacious, and difficult to identify. Typical applications of rootkits include key loggers that collect passwords, utilities to conceal any malware, network traffic sniffers, utilities to gain control of zombie machines and devise other attacks such as denial-of-service, email spamming, etc. Note that all these applications run in a stealthy mode and the system administrator will be fooled with an illusion of maintaining a clean system. As shown in recent McAfee reports, there has been an exponential growth in the number of rootkit techniques, rootkits will conceal an overwhelming (84%) majority of malicious code by 2008 [5]. After an initial kernel-level attack, the rootkit installs itself and conceals all the spawned malware processes without leaving any trace for the system administrators and software anti-malware tools. Rootkits achieve this by manipulating the compromised kernel to hijack all the utilities used by system administrators. Once this attack succeeds, malware applications will be free to exploit the entire system with no limit. The most common objective of a rootkit is to run malware applications on the compromised machine in stealth to (illegally) collect private, vital information.[1]

A number of researchers have proposed techniques to address the rootkit issues at the OS and virtual machine monitor levels [15, 32]. However, these software-based solutions can be easily circumvented and are still susceptible to certain attacks. As mentioned earlier, this type of approach combats the problem at the same privilege level with the kernel rootkits, which is ineffective for solving the issue once and for all. For example, detecting virtualization-based rootkits [26] that work at the hypervisor level is impossible using software anti-rootkit solutions. Addressing these shortcomings necessitates architectural support to provide a direct feedback path between the hardware and the system administrator, bypassing the compromised kernel. In this paper, we propose a process context-aware architecture, called SHARK, that can identify process contexts that are utilizing hardware resources without the OS' decree. By making use of such an architecture, system administrators can directly examine the feedback provided by the underlying hardware and compare it against the OS retrieved data to know the state of the un-trusted OS. In this work, we focus on techniques to reveal malware applications, malware virtual machines and hypervisors running in stealth. To the best of our knowledge, this is the first effort using a synergistic microarchitecture and OS technique to address the rootkit exploits.

The rest of this paper is organized as follows. Section 2 overviews rootkits, the nature of stealth, the existing anti-rootkit solutions. Section 3 introduces the solution space exploration we performed. Section 4 details the proposed architecture, SHARK, and Section 5 discusses the implementation details and analyzes our experimen-

[1]It is debatable whether it is legal or not for managers to install stealth adware to monitor their employees' web-browsing history.

tal results. Section 6 talks about the related work and Section 7 provides the conclusion.

2. ROOTKITS AND STEALTH

Rootkit is a program or a set of programs used by adversaries to hide their presence after a successful, initial exploit. The rootkit's main purpose is to hide processes, files, network connections, and registry entries used by malware from the system administrators' utilities. Malware uses rootkits as an enabler to hide its existence on the machine while abusing all the hardware resources. Rootkits are of two types — memory-based rootkits and persistent rootkits. Memory-based rootkits cannot survive a system reboot because they operate only on the system memory, but persistent rootkits change the persistent data to load itself during a system reboot. It is easier for anti-rootkit tools to catch persistent rootkits by checking the integrity of critical disk data before shutting down the machine. Rootkits are receiving more attention these days as they are becoming serious security threats to all classes of computing, including embedded devices, desktop users, and server farm machines. A notorious example is the Sony rootkit incident of 2005 [6], in which Sony installed hidden software to spy on end-users' playback activities without their consent.

In the next few sections, we classify different types of rootkits, followed by an overview of existing anti-rootkit techniques today and their insufficiency for dealing with increasingly sophisticated rootkits.

2.1 Common Exploit Techniques by Rootkits

Rootkits modify the OS execution flow and data to hide malware processes, net connections, and files from the utilities for detecting suspicious activity. Depending on the level of exploitation, a rootkit can operate in the user space and the kernel space. *Kernel mode rootkits* are more detrimental than *user mode rootkits* as they can obtain unrestricted accesses at the root privilege level and thus can freely manipulate any component of the system via the compromised OS. The following techniques are commonly used by kernel rootkits to achieve malware's stealth and subvert the targeted system:

- **System Service Descriptor Table Hooks (SSDT):** SSDT, also known as the system call table, is the kernel table containing function pointers to handle system calls. A kernel mode rootkit can modify the SSDT entries, replace a function pointer with an address of its own, and hijack the system. This is a simple and popular type of attack accomplished by any Loadable Kernel Module (LKM). As all the utilities used by the administrators perform system calls to obtain the system state, it is easy for the rootkit to intercept these calls and compromise the data confidentiality.
- **Interrupt Descriptor Table Hooks (IDT):** IDT is another table used for storing the interrupt handlers in the kernel. The kernel mode rootkit can replace a legitimate interrupt handler with the rootkit's fake handler. This technique is used by keylogging malware that intercepts keystrokes of interest, e.g., passwords, social security numbers, banking accounts, without any knowledge of the user.
- **Direct Kernel Object Modification (DKOM):** With the DKOM technique, the rootkit will modify some OS data objects directly and remove the information pertaining to the processes the malware intends to hide. For example, the rootkit can delete elements that correspond to the malware's processes from a certain linked list maintained by the OS to represent active processes on the system. The utility tools of the system administrators only see this modified linked list and will not observe any sign of unintended use of computing resources.[2] This technique is hard to detect because it is very difficult to track changes in the OS data. Although people have proposed software tools like Klister [21] to identify processes by reading critical data-structures maintained by the scheduler that cannot be modified, these tools still cannot track changes to the kernel data. Also, the assumption with Klister is that the list of processes maintained by the scheduler is complete and should include all malware processes. This assumption is flawed, as critical scheduler data structures can delicately be manipulated by removing malware processes from the linked list prior to Klister tool comparison and adding back to the list prior to scheduling these malware processes.

[2]Note that this linked list is maintained separately from the one used by the OS to schedule processes. Otherwise, the malware's process will not get CPU time for execution.

2.2 Sophisticated Rootkits

In a technique called *Virtual Memory Subversion*, the Shadow Walker rootkit [34] fakes memory reads from the integrity-checking utilities. Typically, integrity-checking utilities read memory contents periodically to indicate any modification by malware. This rootkit tries to return the original legitimate data when it detects that some integrity checker is accessing the pages. When it reaches its time to execute, the malware uses modified contents of the memory. To accomplish this, the pages used by malware are marked as non-present in the page table and the page fault handler is hooked to intercept accesses to malware's pages. The rootkit differentiates between memory read and execute operations in the handler, returns the original legitimate data if it is a non-execute memory access or modified contents of the page where the malware code is resident if it is a memory-execute operation.

A complex rootkit called *SubVirt* [20] was recently demonstrated. It can be installed as a virtual machine monitor beneath the host OS, making it a guest OS. As the rootkit operates beneath the host OS, it cannot be detected by the host. Also, this Virtual Machine Based Rootkit (VMBR) installs other guest malware OSs that are protected from the original host and run many malware applications completely isolated from the original host OS. Another conceptual rootkit called *Blue Pill* [30] makes use of the advanced hardware-assisted secure virtual machine (SVM) support in the AMD-V technology and is claimed to be similarly applicable to Intel's VT-x technology. Unlike SubVirt, BluePill is capable of installing a thin hypervisor *on-the-fly* and downgrades the host OS to become a guest OS. It becomes extremely challenging to detect them using any off-line detection mechanism. In [26], Rutkowska showed that today we cannot effectively prove or detect virtualization-based rootkits. In another recent work, Cloaker [11] demonstrated a rootkit that exploits hardware to conceal itself without modifying the OS code and data. One of the hardware configuration registers of an ARM-based platform is modified to change the location of interrupt service routines and a virtualized environment from the host OS is created without modifying the kernel image. Our proposed scheme works at the micro-architectural level and controls all software layers running over the bare hardware. It is effective in identifying hidden contexts in all the layers of software, including the VMM or hypervisor level. More details are described in Section 4.5

2.3 Software Anti-Rootkit Techniques

Software-based anti-rootkit techniques use one of the following techniques to examine the corrupted system:

- **Signature-based detection:** The system memory is scanned to identify a sequence of bytes that form a *fingerprint* that is unique to a particular rootkit. This technique is only effective in detecting rootkits with known signatures.
- **Heuristic/Behavioral detection:** In this approach, deviations in the expected normal system behavior are used to identify potential rootkits. Patchfinder [27] works based on the observation that a rootkit should inject additional, spurious code, which will increase the execution time and the number of instructions. Given this rationale, it makes use of instruction count for rootkit detection. The drawback is that false positives are often gener-

ated due to the complexity of the OS, with many possible execution paths that lead to non-deterministic instruction counts.

- **Cross View-based detection:** This technique compares a low-level system view obtained by low-level OS data and functions with the high-level view obtained by compromised high-level OS calls. Any mismatch in the comparison will help to detect rootkits. Rootkit Revealer [25], Klister [21], Blacklight [8] and StriderGhostbuster [16] use this technique.
- **Integrity-based detection:** This approach compares the current snapshot of system memory with a trusted baseline. Any difference in comparison is taken as evidence of malicious activities. Tripwire and System Virginity Verifier [28] are developed based on this technique.

All the existing software techniques until now are inherently flawed because they are executed together with the same corrupted software stack. This gives rise to an endless battle between the rootkit camp and the anti-rootkit camp, each trying to overtake the other. On the other hand, the increasingly complex rootkits such as SubVirt, Shadow Walker, BluePill make it much more difficult for such software anti-rootkit techniques to be effective.

2.4 Hardware Anti-Rootkit Techniques

The Copilot [24] hardware detection scheme aimed at providing a more reliable, OS-independent hardware solution. It relies on hardware-based RAM acquisition to check the integrity of RAM by a co-processor in an isolated environment inaccessible to the compromised machine. A snapshot of the system memory is sent through the PCI bus to a co-processor where the system's integrity is continuously checked. An attack against this system was demonstrated by Rutkowska in [31]. It creates different views of the system memory to the processor and the PCI device to subvert this hardware solution.

3. EXPLORING ARCHITECTURAL SOLUTIONS

As discussed earlier, none of the existing solutions are strong enough to defend a system against rootkits. Worse yet, detecting hidden VMMs has been claimed to be impossible using software techniques [26], which calls for an OS-independent hardware solution. As stealth is the most common and detrimental exploitation of rootkits, we focus on the stealth execution of contexts achieved by non-persistent kernel rootkits in this work. Viewing the problem from a hardware perspective, if the hardware is given the capability to identify running contexts, it can surely help to solve the problem of stealth. The main goal was to identify processes running on hardware and create a master list of processes in hardware. Motivated by the Cross View-based approach, the next step is to compare this master hardware list with the manipulated list of processes obtained by the compromised OS. Any difference in the lists implies that there are hidden processes in the system. Having many constraints in mind, we investigated different ideas that could be employed and discussed their weaknesses before introducing our final approach.

3.1 Tagged TLB

First, we considered using a tagged TLB that contains the process identifier(PID) information of the running processes. In fact, many processors today employ PID (or ASID) in their TLB to avoid TLB flushing upon context switches. For example, in the MIPS processor, the CP0 register is loaded with the ASID of a process before context switching. Upon a context switch, using PID in CP0 register, a secured hardware list of processes can be exported with the PIDs of all running processes, including the stealth ones. In the beginning, this appeared to be effective in tracking every active process in hardware and in identifying processes in stealth. Unfortunately, this does not prevent an OS, compromised by rootkits, from manipulating the hardware PID list since these PIDs are entered by the faulted OS. The compromised OS can easily hijack a legitimate process's PID to masquerade the malware's true identity before the mapping is loaded from the page table entry to the TLB. Note that using a different PID does not affect the correctness of the malware's execution. To circumvent the falsely matched address translation issues, the compromised OS can invalidate the TLB entries pertaining to the hijacked process or simply flush the entire TLB prior to switching to the malware's process. For example, in the x86 architecture (our target platform in this study), such operations can be done by executing an INVLPG <address> instruction or writing to certain flags of the control register (e.g., PG or PE bit in the CR3).

3.2 Tracking based on PDBA

In x86 architectures, upon each context switch, the page directory base address (PDBA) of the candidate process will be loaded into the CR3 register. Using CR3 makes context switching simpler by simply changing the pointer to the corresponding page table base of the process to be executed. On a TLB miss, a hardware-assisted page table walk uses CR3 to look for the virtual-to-physical address mapping in its own page table. Since each PDBA is unique for a process, one can consider using it to create the hardware list of active processes to detect rootkits.

At first glance, this approach seems to be more secure than the tagged TLB scheme as the malware process cannot use the PDBA of a legitimate process; otherwise it will not execute correctly due to incorrect mappings. The dependency between CR3 and the execution context of the malware process appears to make the security scheme strong. Nevertheless, again, it is not very difficult for a compromised OS to circumvent this. A simple attack is that the OS can swap the page tables between the malware and a victim legitimate process before the malware is ready for execution. Once this is done, the malware process can use the PDBA of the hijacked legitimate process as its page table base, which now points to the malware's page table. As such, the hardware list will not contain the PDBA of the malware's process.

4. SHARK: A PROCESS CONTEXT AWARE ARCHITECTURE

After exploring possible architectural solutions, it is evident that all the shortcomings were due to the tightly coupled dependency of these mechanisms with the OS itself, which could have already been compromised. As such, detecting rootkits with OS's direct intervention will always fail. To address this issue once and for all, we saw the need to design a processor architecture to be process context-aware, i.e., adding minimal support for process management in the hardware. The rationale behind our ideas includes: (1) use the hardware's assistance during the creation of a process, (2) isolate and protect the context information within a hardware-hardened sandbox that cannot be circumvented by a compromised OS, and (3) provide the capability of identifying active processes without any knowledge from the OS. The information about each process can be directly obtained through the hardware without any reliance on a vulnerable software stack.

Toward these objectives, we propose a novel processor architecture called *SHARK*, which stands for *Secure Hardware support Against RootKit*. In a SHARK processor, the master control of processes is delegated to the hardware for enforcing the security of process contexts. The OS simply carries out the regular required operations, e.g., TLB lookup, page translation, etc., under SHARK hardware's supervision and assistance. Figure 1 gives an overview of our proposed system extension, including hardware support and software mechanism, to construct a SHARK processor. The rootkit detection capability is accomplished by integrating *Hardware-Assisted PID Generation, Process Page Table Encryption and Decryption, and Process Authentication* into one processor. These mechanisms are implemented within the SHARK security manager, a hardware-based microarchitectural extension while

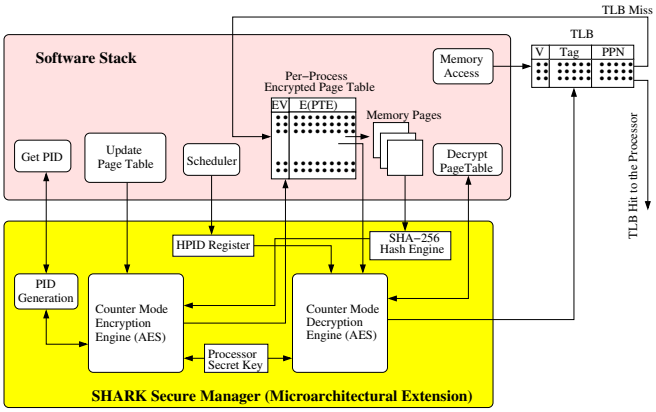


Figure 1: Architectural Support for SHARK Processor

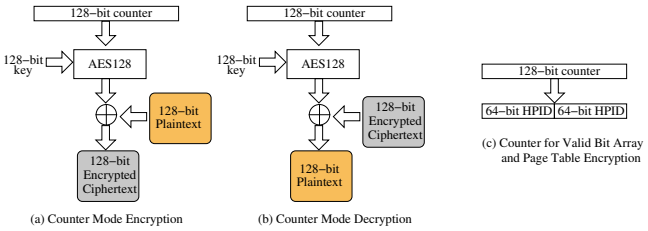


Figure 2: Counter Mode Encryption and Decryption

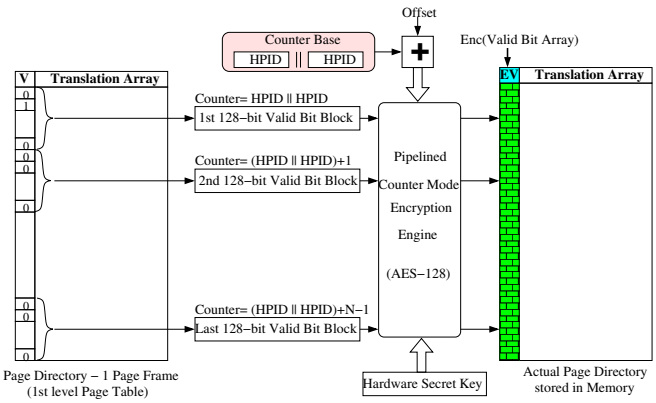


Figure 3: Valid Bit Array Encryption

working seamlessly with the OS (the Software Stack). The following sections detail each component in SHARK.

4.1 Hardware-Assisted PID Generation

Conventionally, the process PIDs are generated and maintained solely by the OS. As discussed earlier, the software-generated PIDs are vulnerable and subject to exploitation by rootkits to conceal malware's identity. Under such schemes, it is impossible to devise a secure mechanism to reveal all active process contexts within the vulnerable software stack. Seeing this deficiency, the first attribute we introduce in SHARK is to perform *Process ID registration* upon a process' creation before it is given permission to make use of any hardware resource. In other words, a PID value will be assigned by the SHARK Security Manager (SSM), part of our microarchitectural extension, rather than by the vulnerable OS. Note that this hardware-generated PID value need not be a secret, as it is simply used as a nonce or a counter value in our counter-mode encryption [12], to be described later. Whenever the process and its page tables are being created, a new 64-bit PID value for the newly created process will be assigned directly by the hardware. By using a 64-bit PID counter and just incrementing the counter for a new process, even if a new process is created every cycle on a 1GHz processor, it takes 584 years to overflow. This is long enough for a system to reboot and initialize the counter. Because of this, the hardware PID generator becomes very simple without any internal buffers and PID pool management logic. The size of the PID value, 64 or 128 bits, will not affect the security strength of our scheme; this will be further explained in the following section. Thereafter, the OS needs to use this value as the PID for this newly created process. For each context switch, the OS will load the PID of the scheduled process into a *Hardware PID (or HPID) register* in the hardware to indicate the running process. In essence, this is similar to loading the page directory base into CR3 in an x86 machine upon context switches. The HPID is an integral part of our proposed encryption/decryption mechanism, described in Section 4.2.

4.2 Process Page Table Encryption and Decryption

The proposed hardware-generated PID and its associated HPID register are not the solution for rootkit prevention or detection. Generating the PIDs in the hardware and asking the OS to load the HPID register with this PID will not solve the problem of stealth, as the untrusted OS can still employ the threat models described in Section 3 and load a hijacked PID to misguide the hardware and conceal malware's presence. What we need is to establish a *dependency* using this HPID between the SHARK hardware and the software stack in such a way that if the software stack, i.e., the compromised OS, attempts to circumvent or break the enforced dependency, the execution of malware's process will be affected. This dependency between the HPID and the execution of the process is achieved via *Process Page Table Encryption*. Using this scheme, the page table of each process kept by the OS will be encrypted in a unique way (proposed below) using counter mode encryption hardware in the SHARK Security Manager illustrated in Figure 1. Before we detail the encryption process, we briefly review the counter mode encryption.

4.2.1 Counter Mode Encryption

Counter mode encryption is a common symmetric-key encryption scheme [12]. It uses a block cipher (e.g., AES [13]), a keyed invertible transform that can be applied to short fixed-length bit strings. To encrypt with the counter mode, one starts with a plaintext, a counter, a block cipher, and a secret key. An encryption key bitstream is generated as shown in Figure 2(a). This key bitstream is XORed with the plaintext bit string, producing the encrypted string ciphertext. To decrypt, the same encryption pad is computed based on the same counter and key, XORs the pad with, and then restores the plaintext, as shown in Figure 2(b).

Counter mode is known to be secure against chosen-plaintext attacks, meaning the ciphertexts hide all partial information about the plaintext, even if some a priori information about the plaintext is known. This has been formally proven in [7]. Security holds under the assumptions that the underlying block cipher is a pseudo-random function family (this is conjectured to be true for AES) and that a new unique counter value is used at every step. Thus a sequence number, a time stamp, or a random number can be used as a counter value. Note that the counter is not a secret and does not have to be encrypted.

4.2.2 Decoupled Valid Bit Array Encryption

When a new process is created, the OS requests the SHARK Security Manager (SSM) to issue a new PID for the process providing the page directory (the first level page table) base address of the respective process. The SSM generates a new PID for the process, as mentioned in Section 4.1, using this PID, encrypts the *valid bit array* of the first level page table (Page Directory) and the newly mapped Page Table Entry (PTE) in the last level page table (i.e., the leaf node). In this section, we describe the valid bit array en-

cryption in the page directory. In Section 4.2.3, we describe the PTE encryption details. First, we implement a hardware secret key that can never be read out by any means similar to what was used in prior secure processors [22, 33, 35]. Using the newly generated PID and this burn-in secret key in the SSM, the first step is to encrypt the entire *valid bit array* in blocks of 128 bits (i.e., every 128 entries) of the first level page table using the encryption engine in the SSM. We propose to concatenate the HPID value of a process, shown in Figure 2(c), as the counter value for encryption. This guarantees a different encrypted valid bit array for each individual process. The counter value will then be incremented by one for each encrypted block starting at 0 for the first 128-bit block.

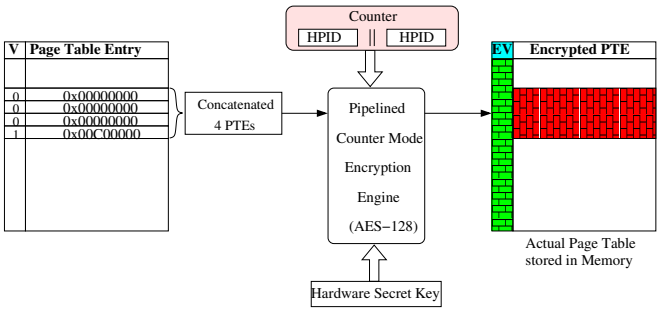
The entire encryption process is illustrated in Figure 3. The result is an encrypted bitstream stored in the original valid bit structure of a page table. Once the valid bit array is encrypted, the SSM returns the hardware-generated PID to the OS for future execution. With regard to the initial valid bit array encryption overhead, for every new process that is created, one 4KB page is allocated as the first level page table. This adds 4KB/128 (input block size for AES) * 80ns = 2560ns in a worst case scenario without using a pipelined AES implementation.[3] This one-time overhead is almost negligible compared to the lifetime of a process. The significance of this initial valid bit array encryption is that it is how we establish the *dependency* or *trust* needed between the SSM and software stack using the HPID and the hardware key. For any subsequent page table walks to be valid, the OS cannot hijack a fake PID, as the same PID must be used to decrypt the valid array prior to obtaining or modifying page allocation information. The importance of valid-bit array encryption will be explained again in Section 4.5.

4.2.3 Page Table Translation Encryption and Updates

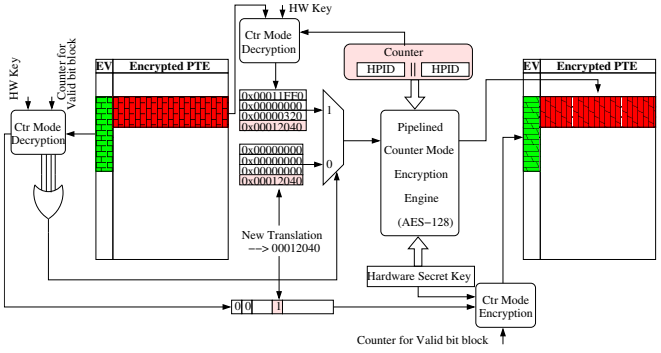
Second, the translation (i.e., the content) of the last level page table (PTE) is also encrypted. Again, we employ the counter mode encryption and use a counter value as shown in Figure 2(c). Similar to the counters used in Section 4.2.2, there is no memory overhead or extra lookup logic involved for storing and searching the counter values, as they are generated at runtime using process context information. The encryption granularity of translations depends on the maximum physical memory that can be supported in an architecture. For example, the current maximal physical address supported for the x86-64 architecture is 40 bits when PAE is enabled in IA-32e mode [18]. Given a 4KB page, the physical base address of the page will be 28 bits. According to the AES standard, the standard block size is 128 bits. Hence, we propose to encrypt four consecutive page table entries (PTEs) at once. It also implies that we may need to pad 4-bit null data for each physical address translation to make it up to 32-bit, which appears to incur 13% memory space overhead for each allocated page table entry. However, it is unconventional to declare a PTE using 28 bits. In fact, the current Linux declares 32 bits (a 4-byte integer) for each PTE; therefore, there is no overhead at all introduced by our scheme in an actual implementation. Figure 4(a) shows the encryption process.

When the OS needs to update the page table of a process, additional work needs to be performed within the page fault handling mechanism. Since the page table is encrypted, to insert a translation into the PTE, the SSM first decrypts the corresponding 128-bit valid bit block in the page directory using the corresponding HPID and performs a hardware page table walk to reach the last level page table. Next, the SSM will examine the four PTEs that contain the target translation. If none of them is valid, the SSM can simply set

[3]Our 80ns is based on our own synthesized AES-128 design. In fact, this assumption is much slower than what was reported in a more recent AES engine implementation fabricated using a 0.18μm process in [17]. This AES coprocessor, a non-pipelined implementation, has an AES-128 encryption datapath running at 330MHz. The total number of rounds is 11; each round can be finished in one coprocessor cycle. With a pipelined implementation, the overall encryption latency will be shortened.



(a) Encryption Process



(b) Decryption Process

Figure 4: Page Table Update in SHARK

the valid bit, re-encrypt the 128-bit valid bit block, and encrypt the assigned PTE with its neighboring three invalid mappings. However, if any of the other three are valid, this implies that the 128-bit encrypted PTE (EPTE) contains some valid address mappings. To maintain correctness, the SSM has to retrieve and decrypt the EPTE, add the new translation, and re-encrypt four PTEs to insert the new translation successfully. Figure 4(b) details the procedures. This process is required for every page table entry update request from the OS. In other words, all the page table updates by the OS will completely be managed by the hardware-based SSM.

4.2.4 SSM-managed TLB updates

During a TLB miss, the TLB needs to be refilled by a hardware page table walk in memory. We target the widely distributed machines such as x86 and PowerPC that use hardware-managed TLBs and build our security module on top of them. We assume that the TLB cannot be tampered with and stores plaintext translation like a regular TLB. Since the page table is encrypted in SHARK, the mappings must be decrypted when performing a page table walk and refilling the TLB. Using the counter values generated from this process context, the SSM will first decrypt the corresponding 128-bit valid bit block in the page directory, performs a hardware page table walk to reach the last level page table, decrypts the corresponding 128-bit valid bit array of the last level page table, and then decrypts the EPTE to obtain the correct decrypted mapping, which is loaded into the TLB. Again, the right HPID must be used to perform the decryption correctly. This makes it compulsory for the compromised OS to load the PID of the malware's process and has to reveal its PID to the hardware whenever it executes. Figure 5 depicts this process for a TLB update.

4.2.5 Instructions supported in SHARK

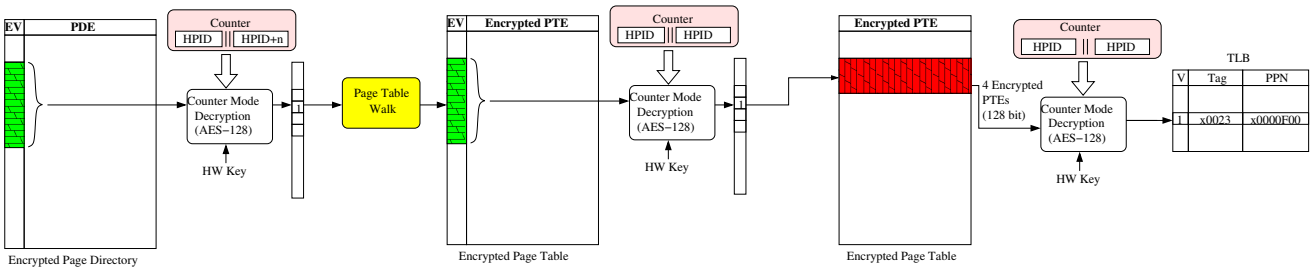


Figure 5: TLB update handled by the SSM

Table 1: Privilege Instruction Support in SHARK

Instruction	Definition	Functions
GENPID	Generate a new PID	Initial Valid-bit array and PTE encryption is performed, M-bit of the respective PTE is set, hardware generated PID is returned
MODPT	Update the page table of a process	Useful when the kernel directly updates page tables of processes (a) If the page is swapped-in for the first time (M-bit = 0), it sets the M-bit and updates the PTE with the new mapping (b) If M-bit = 1 and MODPT is used to invalidate a memory page (swap-out), SHA-256 hash of the memory page is computed and encrypted before swapping the page out (b) If M-bit = 1 and MODPT is used to validate a memory page (swap-in), SHA-256 hash of the memory page is computed, encrypted and compared with the stored encrypted hash to check whether the memory page is owned by the process before updating the PTE
DECPT	Decrypt a process' page table entry.	Useful if the kernel needs to know the physical addresses by directly reading the page tables

Three new privilege instructions supported in SHARK are listed in Table 1. The GENPID instruction is used by the OS when the first memory page is assigned to a process' address space. As mentioned in Section 4.2.2, the PID for each new process is generated by the SSM and the same PID is used to encrypt the following: (1) valid bit array of the page directory, (2) valid bit array of the last level page table, and (3) page table translation (VPN-to-PPN) in the last level page table. The MODPT instruction is required by the virtual memory management module to directly update the page table every time the kernel swaps out and swaps in memory pages. MODPT first decrypts the encrypted PTE, updates the PTE with the new mapping, and then re-encrypts back. In addition, if we are invalidating the PTE (e.g., when swapping a page out), MODPT needs to compute the SHA-256 [4] checksum of the memory page and stores the encrypted checksum before swapping the page out. This is illustrated in Figure 6(a). The resulting 256-bit cipher text accounts for 0.8% memory space overhead for every 4KB page, to prevent a type of attack which hijacks a legitimate process's page table and deliberately remaps malware's pages to PPNs of the hijacked page table described in Section 4.5. This operation is designed to further strengthen the association of a memory page and its legitimate owning process. If MODPT is used to validate a previously mapped page table entry (swapping a page in), before updating the PTE, it checks whether the memory page is owned by the process [4]. This authentication is achieved by computing the checksum of the memory page, encrypting the checksum, and comparing it with the stored encrypted checksum. If the checksums match, this implies that the memory page is truly owned by the process. After this authentication, the PTE mapping in the page table is modified. This is illustrated in Figure 6(b). Note that because of page table encryption and SHA-256 checksum encryption, the malware process cannot modify a PTE of a legitimate process and map it to a malware's page to conceal its activity. The DECPT instruction is required if the kernel wants to read the page table contents directly. This is again useful for page management in the kernel.

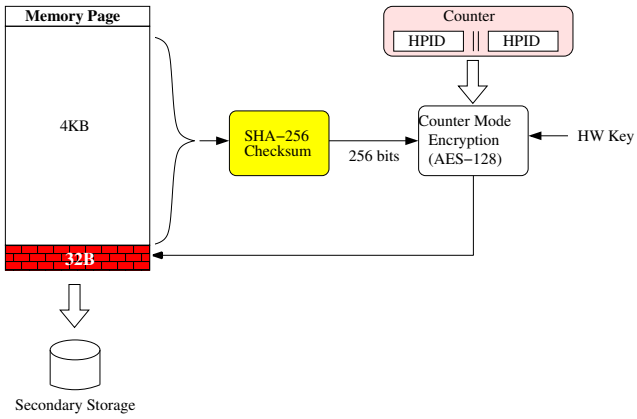
[4]Note that this check is not performed while swapping-in the memory page for the first time. MAPPED bit (M-bit) in every PTE keeps track of this and cannot be modified by OS and is completely managed by SSM. Also note that there is no extra memory overhead to have this extra bit, as this is present in the today's Linux kernel implementation.

4.3 Process Authentication

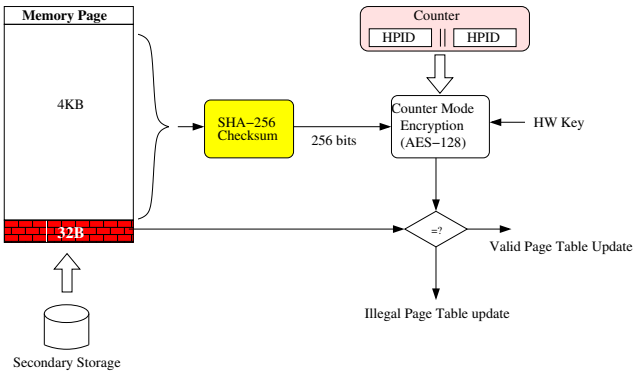
In the SHARK processor, if the compromised OS tries to load the HPID with a hijacked PID from a different process, the decryption will result in an incorrect physical page frame number, which if used, will prevent the malware from executing. The proposed page table encryption and decryption are novel ideas that offer one more level of virtualization provided by the secure hardware for the OS, putting all processes under examination by SHARK. From a security standpoint, this consolidates the binding between the hardware and OS, giving the hardware the capability of controlling and authenticating the execution of software contexts through the address translation process. The whole scheme of PID generation in hardware, page table encryption based on this PID, and decryption based on the same PID in hardware enables the system to perform *Process Authentication*.

4.4 Stealth Checker

The last component of SHARK is the *Stealth Checker*. With the SHARK hardware support, we now have the information of every running process controlled and revealed by the hardware. The hardware extensions provided by Shark Security Manager will make sure that the hardware is not fooled by the OS. But it does not prevent OS from manipulating software utilities like "ps" and "top" to hide malware processes from system administrators. The final action for rootkit detection is to compare the golden list of processes revealed by hardware with the software list returned by the tampered utilities and evaluate the differences. This functionality is implemented by Stealth Checker and it triggers an alarm to the system administrator when the information revealed by software utilities like "ps" and "top" is not consistent with the information from bare hardware. For this operation to be secure, tamper-free,



(a) Swapping Out Memory Pages



(b) Swapping In Memory Pages

Figure 6: Security Enhancement for Using MODPT Instruction

and effective, again the system has to prevent the compromised OS from intervening. Designing a trusted passage between the hardware and anti-rootkit software is very crucial for identifying processes running in stealth.

The stealth checker is implemented in firmware and is called prior to each context switch. Every write to the HPID register triggers this exception. The exception handler in the firmware, reads the HPID register to obtain the PID of the upcoming process. There are no security implications up to this stage because everything is controlled by the hardware or firmware. Even though the firmware can be upgraded by the OS, it requires a system restart that clears out the memory-based rootkits that we target. After the HPID read, the PID with previously buffered PIDs are encrypted and sent to a remote system administrator for examination. The data will be 128B to accommodate 64-bit PIDs of 16 processes. To reduce the network activity, the firmware can send this data once in every 10 context switches. Unlike the prior CoPilot technique, which sends memory pages for examination unprotectedly, our firmware encrypts this data using a 128-bit key given by the system administrator upon a clean firmware installation. Note that the OS cannot compromise these encrypted PIDs even though our firmware uses the potentially compromised OS to send these PIDs over an insecure channel. We also randomized the PID order sent by the firmware to confuse the OS in case the OS can keep track of the sent PIDs and then block the communication containing the malware's PID. To make it more secure, we employ sequence numbers in the packets sent to the remote machine to track if the OS has blocked any packet. If the OS attempts to block or replay the packets sent by the handler, the system administrator can conclude that the OS is compromised and take appropriate actions. Once the remote machine receives these encrypted PIDs, it can decrypt and create the golden list of processes. This list is an event log registered in the remote machine, which can be examined by the system administrator. He can use ssh to remotely connect to the suspected machines and execute "ps" like commands to check the process list returned by their OS. This procedure can be completely automated. Any mismatch will alert one to a probable security breach.

The major sources of overhead for the exception handler are (1) time taken by the kernel network stack to update NIC buffers, and (2) network bandwidth utilized. The minimum time slice in the Linux kernel 2.6 is 5ms, the maximum is 800ms and the average being 100ms. Taking into account the maximum context switching frequency, we have to send 128B data over the network every 50ms (once in 10 context switches). Based on our measurements, the average kernel TCP stack overhead to send 128B is less than 0.1 ms and the network bandwidth utilized is negligible.So the overall overhead of Stealth Checker is less than 0.2% and is highly practical.

4.5 Strength of SHARK

This section discusses potential threat models and analyzes the strength of our SHARK processor. Knowing that the OS cannot be trusted, we have thought about the following mechanisms that might be used by future exploits to subvert SHARK. As we will show, SHARK can prevent all these malicious attempts carried out by an untrusted OS.

First, rootkits could hijack a legitimate process' PID instead of its own to conceal the PID of the malware process. As analyzed previously, this will result in the failure of malware process execution, as it cannot decrypt the address mappings correctly using a different PID. Note that the encryption is seamless, established using the HPID when the process and its page directory are created.

Another attack is that a rootkit may plan to encrypt the page tables of a malware process using the hijacked PID of a legitimate process. This attack will also fail because the page tables are encrypted using the HPID generated by the hardware SSM in the very beginning before the PID value is revealed to the process. As the valid bit array of the page table is also encrypted based on the HPID, it must be decrypted prior to any page table update. This initial decryption of the valid bit array will fail and confuse the address mapping of the malware if the malware uses a hijacked PID.

Now, we will talk about the significance of valid-bit array encryption of the first level page table. If we do not encrypt the first level page table, and just encrypt the last level translations, one attack model can successfully break the defense mechanism of SHARK and use a legitimate process' PID for malware's execution. The attack model is described here- We know that the last level page tables are constructed on-demand, depending on the memory footprint of the application. When the second last level page table is constructed, the contents will not be encrypted by SSM and the OS can use MODPT instruction to encrypt the contents based on a legitimate process' PID. From this point, all the subsequent translations, will be encrypted based on the other process' PID. This will result in just one last level page table, encrypted using malware's PID and the rest encrypted based on some legitimate process' PID. If the malware application uses page tables other than the first one (encrypted based on malware's PID), it can successfully execute by using the legitimate process' PID. To defend against this attack, we encrypt the root node of the page table (first level) so that, all the subsequent modifications (on-demand construction of last level page table) should be first authenticated by the successful decryption of the first level page table. This makes sure that malware's page tables are not constructed using other legitimate process' PID.

Yet another attack model is to have the malware invalidate all of the allocated pages and swap all malware process' pages to the disk.

Then the malware will start over and encrypt the blank page table using a hijacked PID of a legitimate process before it is brought back to the memory. This is not possible, again, due to the separate encryption of the valid bit array. The PTE invalidation will also cause page table updates that will subsequently encrypt the valid bits of the page table. Even if the pages are swapped out, the page table will still have valid bits encrypted and the hardware page walk mechanism will exercise the SSM-enforced decryption for invoking page faults. If they are not decrypted and re-encrypted correctly, the page table will never be updated properly.

One may wonder why the compromised OS cannot simply update the page table with its own encrypted valid bit array and translation using hijacked PID since the page tables are all in memory. This is impossible in SHARK since the counter mode encryption relies on a hardware burn-in secret key, which can never be read out by any means. It is hardwired into the AES engine for performing encryption and decryption. Therefore, this threat model is not feasible, either.

Another attack could manipulate a legitimate process' page table and address space to run malware. Two types of this attack could be launched: (1) Using the MODPT instruction, modify a duplicate copy of a legitimate process' page table to map to malware's physical pages. Note that the OS has all the information required — physical pages used for malware and legitimate processes' decrypted page table structures; (2) Use legitimate process' address space to run malware— swapping malware code and data to legitimate process' memory pages and using manipulated legitimate process' page tables to run malware. Note that this attack is an extreme strategy to hide malware and is very difficult to achieve. Even if the above attacks can be somehow devised by malware, SHARK will be successful in defending against them. This is achieved by the SHA-256 checksum mechanism described in Section 4.2.5 and its encryption, which gives SSM the capability to track the ownership of memory pages. This will not allow the OS to manipulate the page tables to point to memory pages used by other processes on-the-fly or take other process' PID and use its memory pages while it is still executing.

Last but not least, we know that virtual machine-based rootkits [20, 30, 26] are emerging these days; we discuss the implications of SHARK scheme against these rootkits here. In virtual machine-based rootkits, the malicious software uses either hardware virtualization support or changes the boot sequence to load itself as a VMM under the host OS. Once the host OS starts operating above a layer of malicious VMM, it is completely compromised. Now let us concentrate on the unsolved problem of identifying the nested VMMs installed using hardware virtualization support in [26]. By using SHARK, we can effectively combat the problem of identifying these hidden virtual machines. Private page tables, shadow page tables and nested page tables using hardware technology in AMD processors are the techniques used by BluePill malware to hide the malware VMMs in memory. By using SHARK hardware, the new page tables created in the hypervisor must be registered to obtain a key and then pass through SSM process authentication before executing these contexts. Using this technique, even if the malware is able to hide its page tables from the host OS and integrity checking tools, it cannot fake its identity to the hardware. In this way, the proposed SSM has control over the VMMs, too. The PIDs of contexts inside VMMs are logged continuously in hardware and hence the execution of unintended VMMs is revealed to the system administrator.

5. EXPERIMENTAL ANALYSIS

We conducted two sets of experiments to evaluate the proposed SHARK Security Manager. First, we evaluated the practicality and strength of the proposed scheme against malware running in stealth using real kernel rootkits available on Linux. Following that, we performed performance experiments to quantify the overheads incurred by the SHARK architecture.

5.1 Functionality Evaluation

As a proof of concept, we installed several rootkits on a Linux OS running on top of an emulated SHARK processor. Bochs, a highly portable open source x86 PC emulator, was used to emulate the entire system, including the SHARK security manager and the infected OS.

To verify that the proposed scheme is practical, memory management unit, process management unit, and scheduler of the Linux kernel versions 2.2.14 and 2.6.16.33 were modified and recompiled to use the SSM implemented in Bochs to support our proposed mechanism. Using the new instructions (shown in Table 1) supported by SHARK, the kernel was modified. Note that these instructions are safe to use, because page tables are not updated or decrypted correctly if a different HPID value is used. The modified kernel boots and executes all the processes perfectly with encrypted page tables. To support pointers to kernel page tables in all user page tables, on a TLB miss, we differentiate between kernel space and user space memory accesses to use the appropriate key for decryption. Shared libraries is not an issue because even if two mappings of different processes take you to the same physical frame, different PIDs will be used to encrypt their respective page tables. As the kernel has a fixed range of virtual addresses for these shared pages, SHA-256 hash is not maintained and checked for these pages. In the case of virtualization systems, the lowest layer, i.e., the hypervisor, must use the ISA support provided by SHARK.

To evaluate security, we installed rootkits over the modified kernel for SHARK running over the emulated SHARK hardware architecture. The following five rootkits collected from [3] were inserted into the base kernel as Loadable Kernel Modules (LKMs): *Adore 0.42*, *Knark 2.4.3*, *Phide*, *Enyelkm.en.v1.1*, and *Mood-nt-2.3*. Note that, not all these rootkits are compatible with different Linux kernels. The first three rootkits were aimed at the Linux 2.2 kernel, while the last two were developed particularly for subverting the Linux 2.6 kernel. These rootkits inserted as LKMs can access the kernel space and will modify the system call table, interrupt descriptor table to alter the execution flow of the compromised OS, and provide utilities to conceal malware's processes from the system administrator's utilities (e.g., ps, top). Under such a setup, we contrived a compromised software stack to be able to assess the effectiveness of our SHARK architecture. The scheduler of the base kernel was modified to load the HPID register with the PID of the process prior to a context switch. Every write to this HPID register results in an exception serviced by the SSM. As described in Section 4.4, this exception handler resides in the firmware and cannot be tampered by the compromised OS. In this way, the PIDs of running processes are read by the firmware and a golden list of processes is created. The compromised utilities such as ps or top were queried to obtain a list of processes. As these utilities were compromised, the information of malware processes were concealed. By comparing the two lists, hidden malware processes were detected. For all these functional experiments, our SSM succeeded in all scenarios by triggering security alarms to reveal the processes running in stealth, demonstrating the effectiveness of our SHARK architecture.

5.2 Performance Evaluation

Since SHARK introduced extra encryption/decryption overhead to protect the process page tables, we now evaluate the overhead's impact on performance. We obtained cycle information from Virtutech's Simics [23] with its gcache model enabled.[5] A staller will stall the cycle accounting mechanism whenever a cache miss occurs. Since Simics does not provide an out-of-order cycle-level single-processor model and the staller essentially implements a blocking cache, our overhead estimation may be somewhat pessimistic.

[5]The reason we did not use Simics for functionality evaluation is due in part to the fact that some modules, e.g., Page Table Walks, in Simics are not open-sourced; thus we could not modify them to model our SHARK implementation.

Table 2: Processor System Configurations

Configuration	freq	L1	L2	AES latency	SHA-256 latency	Memory latency
Config1	2GHz	32KB, 8-way, 64B line 2 cycles	2MB, 12 cycles 8-way, 64B line	80	138	200
Config2				160		
Config3			4MB, 19 cycles 16-way, 64B line	80		
Config4				160		
Config5	4GHz	32KB, 8-way, 64B line 3 cycles	2MB, 25 cycles 8-way, 64B line	160	276	300
Config6				240		
Config7			4MB, 38 cycles 16-way, 64B line	160		
Config8				240		

To model a modern processor, we chose our cache and TLB configurations to closely resemble those in the Core microarchitecture such as Conroe core from Intel. The cache access times were estimated based on Cacti 4.2 [2] with two target frequencies specified in Table 2. We assume there are two read/write ports for L1 and one unified read/write port and one snoop read port for the L2. Note that, x86 ISA supports mixed page sizes; thus there are two TLBs for two different page sizes: 4KB and 2MB, used for each machine. We also varied the number of TLB entries to study their sensitivity. Furthermore, we studied the sensitivity of the AES engine latency. We assume that a baseline 10-round AES-128 takes 80 cycles on a 2GHz processor, similar to an optimized design reported in [19]. Then we increase the latency for different machine configurations. We assume that a baseline pipelined SHA-256 hashing engine takes 138 cycles on 2GHz processor, similar to the implementation in [10]. Then we increase this latency for the faster processor. The entire configurations are listed in Table 2.

We chose 28 SPEC 2006 programs as our benchmark, using a reference input set. For each simulation, we emulated the first two billion instructions including instructions from the OS code. We did not fast-forward instructions, in order to obtain more page faults and TLB updates. In reality, the overall overhead should be much smaller. Linux kernel 2.6.16.33 was recompiled to send requests to the SHARK security manager whenever a page table update and PTE decryption were needed. When the SSM gets a request to update PTE, it encrypts the PTE and updates the page table. This requires one valid bit array decryption + one PTE decryption + one PTE re-encryption + one valid bit array encryption. Also on every page table update, we need to compute SHA-256 hash of the 4KB page and encrypt the 32B hash. This adds an overhead of SHA-256 hashing latency + two AES Encryptions. The overall overhead for a page table update will be six times the AES latency + SHA-256 hashing latency. Also, we have to decrypt the the corresponding PTE for each TLB refill. This requires two valid bit array decryptions + one PTE decryption. A TLB miss to handle hardware page table walk is conservatively assumed to be 30 cycles. More penalty in the baseline TLB miss will dilute our overhead. The actual page faults are handled by the OS code and these explicit OS instructions were accounted for in the emulation. The page table updates, page table decryptions, and the TLB updates account for the sources of overhead. Also, we need to flush the TLB upon every context switch as in x86 machines.

Figure 7 shows the cycle time overhead for all the benchmark programs running with six different TLB organizations using Processor Config1. We observed that TLB organizations are critical to the overheads. Obviously, some benchmark programs such as 401.bzip2, 410.bwaves, 459.GemsFDTD, 470.lbm, 998.rand, and 999.rand require more than 2MB page mappings in the TLB. For these applications, when we increased the number of entries in the large TLB (for 2MB page) from eight to 32, the overhead was drastically reduced below 1%. To gain further insight, Figure 8 shows the number of data TLB updates for TLB Config1 and TLB Config2.[6] The only difference between these two configurations is the number of TLB entries for 2MB pages. It is evident that the same benchmark programs show a huge reduction in the d-TLB updates when more translation entries are employed in the 2MB-page TLB.

In Figure 7, 401.bzip2, 410.bwaves, and 470.lbm also demonstrate higher overhead than the others. This can be explained by examining the context switch frequencies shown in Figure 9. These three show a much higher number of context switches, a few orders or magnitude higher than the others. As the TLBs are flushed during each context switch, we will need to refill the TLB more often, causing the extra overheads in decryption.

Finally, in Figure 10, we show the average overhead for all benchmarks across the eight processor system configurations described in Table 2 with six TLB organizations. For the same generation processor, moving to a larger L2 cache tends to lower the overhead (e.g., Config1 vs. Config3). This is because the longer L2 latency for a larger cache penalizes the baseline and shrinks the overhead proportionally. In general, SHARK merely introduced 4.7% overhead in the worse case, and the overhead is below 1% when a larger TLB (e.g., 4-way, 256 entries) is used.

6. RELATED WORK

Many software techniques have been proposed for kernel mode rootkits detection [9, 27, 36, 29, 28]. These software solutions operate in the same compromised software stack and expect that some kernel components will not be compromised. This assumption is somewhat flawed because the upcoming kernel rootkits can always subvert these trusted components to defeat anti-rootkit solution. Although these software anti-rootkit solutions were dependable when they were released, they are not considered to be secure after release, as the complexity of the rootkits continues to evolve. Coming to the hardware solution proposed, CoPilot, as mentioned in Section 2.4, was proven to be insecure by Joanna Rutkowska in [31]. Also, note that this is not a microarchitectural approach, but rather a system-level solution that was proposed to deal with the problem.

Researchers have proposed using virtual machine monitors (VMMs) to check the integrity of the host OS [14, 32]. These VMMs are typically designed to have minimum code sizes and have a security manager inside the VMM verifying the integrity of the host OS. These techniques are no longer considered to be safe because of rootkits that are exploiting the hardware virtualization support like Blue Pill [30]. In [26], it is shown that, none of the techniques proposed until now can detect virtualization-based rootkits. As SHARK is a microarchitectural solution, it can address these virtual machine-based rootkits effectively, as discussed in Section 4.5.

The microarchitectural research community has dealt with the problem of untrusted OSs [35, 22]. These architectures were proposed to have a secure execution environment without a secured kernel. The main applications that they consider here do not need interactions with the host OS. Their goal is to protect the application's code and data from being tampered with, including the untrusted OS. The attack model that we are considering in this work is different in that the malicious kernel will not try to manipulate the code and data of other applications. Instead, malware uses computing resources stealthily and persists in the system as long as possi-

[6] We found the numbers of i-TLB updates of different TLB sizes almost remain the same.

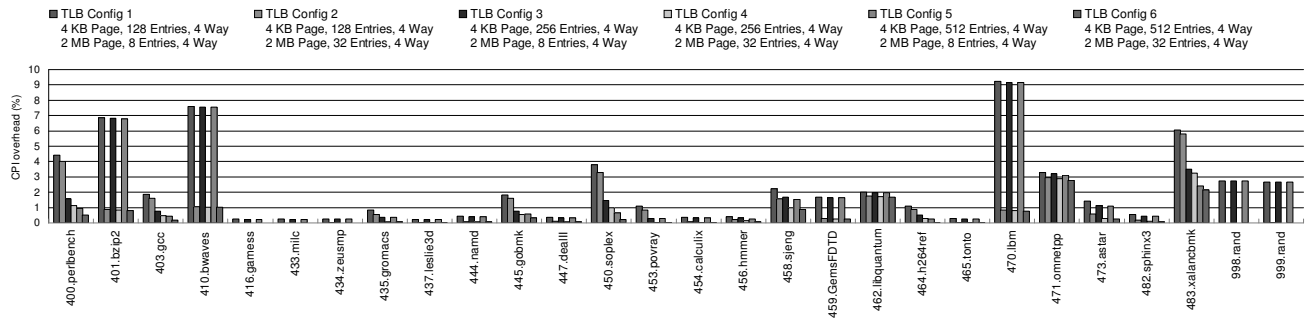


Figure 7: Performance impact with different TLB organizations (Config1)

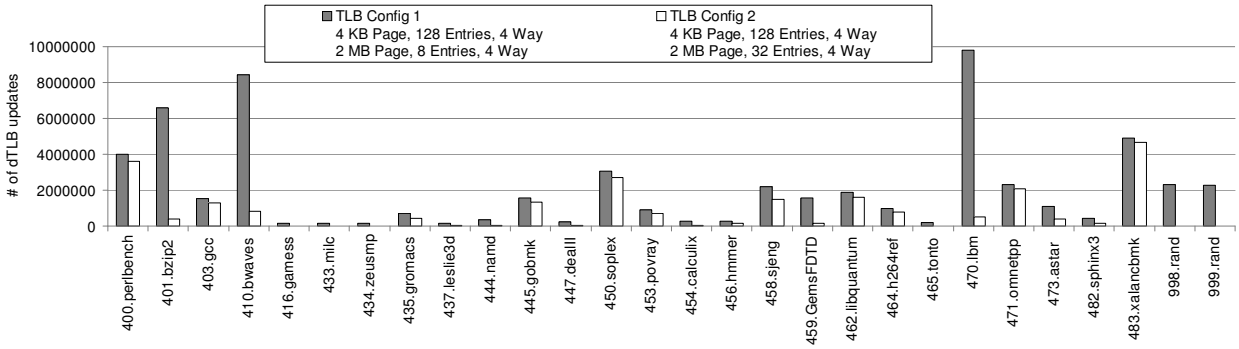


Figure 8: Number of D-TLB updates for TLB Config1 and TLB Config2

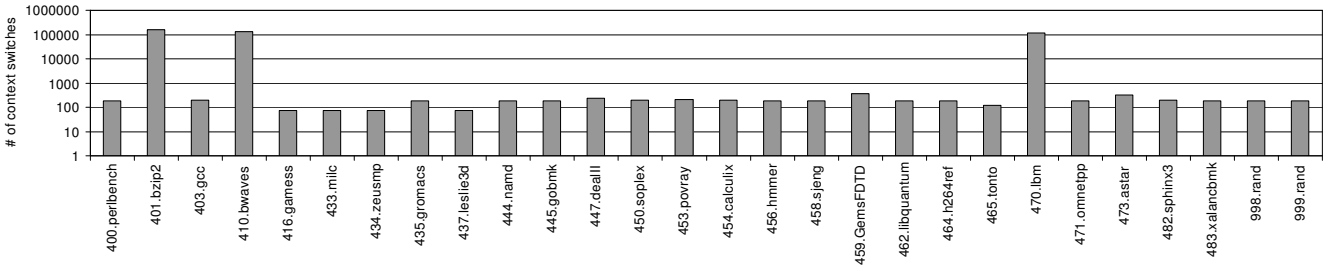


Figure 9: Number of context switches (amid 2 billion instructions)

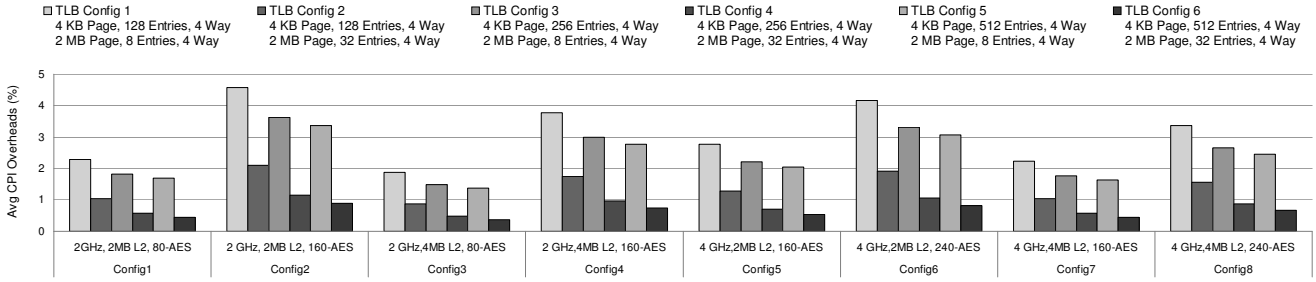


Figure 10: Average overheads for all the benchmarks with different configurations

ble without affecting other applications. In ARM-based TrustZone Technology [1], an isolated on-chip execution environment is made available for security purposes. This design is not a solution for any vulnerability, but a framework that allows one to devise secure systems. This approach is not tightly coupled with the OS, which can cause an endless battle between the secure and non-secure regions. To the best of our knowledge, we are the first to propose microarchitectural support, to enhance the security of OS to deal with applications running in stealth.

7. CONCLUSIONS

Rootkit-based exploits have become a serious concern in cybersecurity. Once a computer is infected, rootkits are detrimental, tenacious, and difficult to identify and remove. Typical applications of rootkits perform key-logging to reveal passwords, sniffing network traffic to steal secrets, and controlling zombie machines to stage other attacks such as email spamming, denial-of-service attacks, etc. They exploit the kernel's vulnerabilities to gain root privileges and continue to run their malware applications on compromised machines. These malware processes operate completely in stealth, leaving no trace for system administrators. To address these issues, this paper proposes an autonomic architecture called SHARK that operates against stealth achieved by rootkits' exploits. To the best of our knowledge, this is the first paper addressing rootkit exploits using a synergistic hardware/system software approach to directly enhance the trust between the hardware and the processes under a compromised OS. SHARK is process context-aware; it employs secure hardware support to provide system-level security, without trusting the software stack, including the OS kernel. The proposed mechanisms, including hardware PID, page table encryption, and process authentication, tightly couple the dependency between the OS and hardware architecture, making the entire system more security-aware. Under SHARK, the concealed malware at user, kernel and VMM levels of the software stack will be revealed automatically by the synergistic cooperation between SHARK and the software stack.

Running Linux OS and installing real-life rootkits, our experimental results show that SHARK is highly effective in identifying rootkits with less than 4.7% performance impact in the worst case and less than 1% performance degradation in typical processor configurations.

8. ACKNOWLEDGMENT

This work was supported in part by an DOE Early CAREER PI Award and an NSF CAREER Award (CNS-0644096).

9. REFERENCES

[1] ARM TrustZone Technology. http://www.arm.com/products/security/trustzone/index.html.

[2] CACTI 4.2, HP Labs. http://hpl.hpl.hp.com/personal/Norman_Jouppi/cacti4.html.

[3] http://packetstormsecurity.org/.

[4] National institute of science and technology fips pub 180-2: Sha256 hashing algorithm.

[5] Rootkits,The Growing Threat, McAfee. http://www.mcafee.com/us/local_content/white_papers/threat_center/wp_akapoor_rootkits1_en.pdf.

[6] Sony BMG CD copy prevention scandal. http://en.wikipedia.org/wiki/2005_Sony_BMG_CD_copy_protection_scandal.

[7] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A concrete security treatment of symmetric encryption. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS '97)*, page 394. IEEE Computer Society, 1997.

[8] Blacklight. http://www.f-secure.com/blacklight.

[9] J. Butler. VICE Catch the hookers. In *www.blackhat.com/presentations/bh-usa-04/bh-us-04-butler/bh-us-04-butler.pdf*, 2004.

[10] L. Dadda, M. Macchetti, and J. Owen. An ASIC design for a high speed implementation of the hash function SHA-256 (384, 512). In *Proceedings of the 14th ACM Great Lakes symposium on VLSI*, 2004.

[11] F. David, E. Chan, J. Carlyle, and R. Campbell. Cloaker: Hardware Supported Rootkit Concealment. In *Proceedings of IEEE Symposium on Security and Privacy, 2008*, 2008.

[12] W. Diffie and M. Hellman. Privacy and Authentication: An Introduction to Cryptography. In *Proceedings of the IEEE*, 1979.

[13] F. I. P. S. Draft. Advanced Encryption Standard (AES). National Institute of Standards and Technology, 2001.

[14] T. Garfinkel. A virtual machine-based platform for trusted computing. In *In Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[15] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 193–206. ACM Press, 2003.

[16] GhostBuster. http://research.microsoft.com/Rootkit/.

[17] D. D. Hwang, K. Tiri, A. Hodjat, B.-C. Lai, S. Yang, P. Schaumont, and I. Verbauwhede. AES-Based Security Coprocessor IC in 0.18μm CMOS with Resistance to Differential Power Analysis Side-Channel Attacks. *IEEE Journal of Solid-State Circuits*, 41(4):781–791, 2006.

[18] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1*, 2007.

[19] T. Kgil, L. Falk, and T. Mudge. Chiplock: support for secure microarchitectures. *SIGARCH Computer Archititecture News*, 33(1):134–143, 2005.

[20] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. SubVirt: Implementing malware with virtual machines. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.

[21] Klister. http://www.rootkit.com/project.php?id=14.

[22] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. B. J. Mitchell, and M. Horowitz. Architectual support for copy and tamper resistant software. In *Proceedings of the 9th Symposium on Architectural Support for Programming Languages and Operating Systems*, 2000.

[23] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, Feb. 2002.

[24] N. Petroni. Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of Usenix Security Symposium*, 2004.

[25] RootkitRevealer. http://technet.microsoft.com/en-us/sysinternals/bb897445.aspx.

[26] J. Rutkowska. Security Challenges in Virtualized Enviroments. http://invisiblethings.org/papers/Security0Enviroments.pdf.

[27] J. Rutkowska. Detecting Windows Server Compromises with Patchfinder 2. In *www.invisiblethings.org/papers/rootkits_detection_with_patchfinder2.pdf*, 2004.

[28] J. Rutkowska. System Virginity Verifier: Defining the Roadmap for Malware Detection on Windows Systems. In *http://www.invisiblethings.org/papers/hitb05_virginity_verifier.ppt*, 2005.

[29] J. Rutkowska. Thoughts about Cross-View based Rootkit Detection. In *http://www.invisiblethings.org/papers/crossview_detection_thoughts.pdf*, 2005.

[30] J. Rutkowska. Introducing the Blue Pill. In *http://theinvisiblethings.blogspot.com/2006/06/introducing-blue-pill.html*, 2006.

[31] J. Rutkowska. Beyond the CPU: Defeating hardware based RAM acquisition. In *In Proceedings of BlackHat DC 2007*, 2007.

[32] A. Seshadri. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *ACM Symposium on Operating Systems Principles*, 2007.

[33] W. Shi, H.-H. S. Lee, M. Ghosh, C. Lu, and A. Boldyreva. High Efficiency Counter Mode Security Architecture via Prediction and Precomputation. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, June 2005.

[34] S. Sparks and J. Butler. Shadow Walker - Raising the bar for Rootkit Detection. In *In Proceedings of BlackHat*, 2005.

[35] E. G. Suh, D. Clarke, M. van Dijk, B. Gassend, and S.Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing . In *Proceedings of the International Conference on Supercomputing*, 2003.

[36] Y.-M. Wang. Detecting Stealth Software with Strider GhostBuster. In *Proceedings of Dependable Systems and Networks*, 2005.