ARCHITECTURAL SUPPORT FOR AUTONOMIC PROTECTION AGAINST STEALTH BY ROOTKIT EXPLOITS

A Thesis
Presented to
The Academic Faculty

by

Vikas R. Vasisht

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
Nov 2008

ARCHITECTURAL SUPPORT FOR AUTONOMIC PROTECTION AGAINST STEALTH BY ROOTKIT EXPLOITS

Approved by:

Professor Hsien-Hsin S. Lee, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Douglas M. Blough
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor John A. Copeland
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: November 14, 2008

To my Parents.

ACKNOWLEDGEMENTS

Without the support from many people, this work would have been impossible. First, I am very grateful for my loving family back home in India – I would like thank my parents and my grandmother, who have always motivated me to perform well and who have supported me emotionally through the hardest times. Also, I would like to thank my loving sister and brother-in-law for guiding me at every stage in the U.S.

Then, I would like to extend my sincere regards to my research advisor, Dr. Hsien-Hsin Sean Lee. He has been an excellent mentor who has guided me throughout my journey at Georgia Tech and I owe him whatever I have achieved in this country. I am really lucky to be his student and receive his extended support during my stay at Georgia Tech.

I would like to thank my fellow colleagues at the MARS lab: Abilash Sekar, Eric Fontaine, Richard Woo, Dong Hyuk Woo, Mrinmoy Ghosh, Dean Lewis, Pratik Marolia, Nak Hee Seong, Ahmad Sharif, Sungkap Yeo and Jen-Cheng Huang, who have inspired me and have helped me to grow professionally.

Last, I would like to thank all my invaluable friends and the great people I have met at Georgia Tech.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Operating system security has become a growing concern these days. As the complexity of software layers increases, the vulnerabilities that can be exploited by adversaries increases. Rootkits are gaining much attention these days in cyber-security. Rootkits are installed by an adversary after he/she gains elevated access to the computer system. Rootkits are used to maintain a consistent undetectable presence in the computer system and help as a toolkit to hide all the malware activities from the system administrator and anti-malware tools. Current defense mechanism used to prevent such activities is to strengthen the OS kernel and fix the known vulnerabilities. Software tools are developed at the OS or virtual machine monitor (VMM) levels to monitor the integrity of the kernel and try to catch any suspicious activity after infection.

Recognizing the failure of software techniques and attempting to solve the endless war between the anti-rootkit and rootkit camps, in this thesis, we propose an autonomic architecture called SHARK, or Secure Hardware support Against RootKits. This new hardware architecture provides system-level security against the stealth activities of rootkits without trusting the entire software stack. It enhances the relationship of the OS and hardware and rules out the possibility of any hidden activity even when the OS is completely compromised. SHARK proposes a novel hardware manager that provides secure association with every software context making use of hardware resources. It helps system administrators to obtain feedback directly from the hardware to reveal all running processes. This direct feedback makes it impossible for rootkits to conceal running software contexts from the system administrator.

We emulated the proposed architecture SHARK by using Bochs hardware simulator and a modified Linux kernel version 2.6.16.33 for the proposed architectural extension. In our emulated environment, we installed several real rootkits to compromise the kernel and concealed malware processes. SHARK is shown to be very effective in defending against a variety of rootkits employing different software schemes. Also, we performed performance analysis using SIMICS simulations and the results show a negligible overhead, making the proposed solution very practical.

CHAPTER I

INTRODUCTION

The security of an operating system directly affects the security of the entire computing system and hence OS security is crucial. Kernel security is becoming critical these days as the complexity of software systems has increased.

Currently, to warrant a safe OS kernel, efforts mainly focus the software, either by changing the architecture of the OS kernel or by fixing the known vulnerabilities. Due to the increasing complexity and the size of the OS, it is unrealistic to design a monolithic OS without any vulnerabilities. On the other hand, researchers have proposed intrusion detection systems (IDS) that aim at periodically monitoring the integrity of critical software components. Although these techniques were sufficient in the past, lately they have proven to be useless for emerging sophisticated attacks. Also, we have recognized that defending against malware in the software stack is not a proactive approach. The software stack is the common battle ground for both malware and malware-detection systems - both trying to counteract each other. In fact, it is a losing battle for software-detection schemes, as it is easy for future malware to devise attacks against the present solution proposed by anti-malware camps, which results in a never-ending loop in providing yet another software solution that will useless in the future. To solve this problem permanently, the correct approach is to make the hardware more security aware and enhance the relationship of the OS and the hardware.

Rootkits are gaining more attention these days, as they are dangerous and difficult to identify.Rootkits are not exploits to gain elevated access to the machine. They are used to hide all the malware activities from the system administrator after an initial

exploit. After an initial exploit, the adversary will gain elevated access to the machine and he/she will gain the freedom to manipulate any software component of the software stack. Rootkits are used to manipulate the operating system to enable hidden malware activity and hide all the processes, network connections and files used by malware. This helps the adversary make use of the computing resources consistently and remain completely hidden from the system administrator. 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. Recent studies indicate that there has been an exponential growth in the number of rootkit techniques, and rootkits will conceal an overwhelming (84%) majority of malware by the end of 2008 [21]. After an initial exploit, the rootkit installs itself and conceals all the malware processes from the system administrator and software anti-malware tools. This is achieved by manipulating the compromised kernel and hijacking all the utilities used by system administrators. The system administrator will be under the illusion of maintaining a clean system by observing a manipulated system state sitting at the top of the corrupted software stack.

Researchers have proposed software techniques to address the rootkit issue at the OS and virtual machine monitor levels [10, 31]. However, software-based solutions can be easily circumvented, as this approach combats the problem at the same privilege level with the kernel rootkits, which is ineffective for solving the issue once and for all. Today, detecting virtualization- based rootkits [25] is a challenge in the security research groups and has proved to be impossible to solve using software techniques. This necessitates a micro-architectural solution to enhance the relationship of the OS and hardware and provide direct feedback to the system administrator to reveal software contexts making use of hardware resources to solve the problem of stealth permanently. In this thesis, we propose SHARK, which stands for Secure Hardware

Against RootKits, a process-context-aware architecture that has the capability to identify every software context making use of hardware resources. When new processes are created, these processes have to go through the secure hardware manager to register themselves and then they always have to go through a process authentication phase before making use of hardware resources. This enables hardware to identify every software context making use of hardware resources at any point in time. The next step is to provide direct feedback to the system administrator and expose the master list of software contexts making use of hardware resources. The system administrator can make use of this master list of processes exposed by hardware, which cannot be manipulated, and compare it with the process listing returned by the OS. If the OS is compromised, and the malware processes are hidden, this comparison will result in a mismatch that can be used to trigger an alarm. As this solution is at the micro-architectural level, any hidden software contexts at any layer of the software stack will be revealed to the system administrator without having to rely on the compromised OS. To the best of our knowledge, this is the first effort that uses a synergistic micro-architecture and OS technique to address rootkit exploits. The rest of the thesis is organized as follows. Chapter 2 gives an overview of rootkits, the nature of stealth, and existing anti-rootkit solutions. Chapter 3 introduces the explored solution space we performed before we proposed the new architecture. Chapter 4 gives all the details of the proposed architecture SHARK, and Chapter 5 discusses the implementation details and the analyzes our experimental results. Chapter 6 talks about the related work, Chapter 7 discusses the future work, and Chapter 8 provides the conclusion.

CHAPTER II

ROOTKITS OVERVIEW

This chapter gives an overview of rootkits, the techniques used by rootkits to achieve stealth in a compromised machine, the software anti-rootkit techniques proposed and their weaknesses, and the emerging and future challenges.

A rootkit is a set of programs used by adversaries to achieve a permanent or consistent undetectable presence on a machine. The attack scenario is as follows: The adversary first uses a known kernel vulnerability to gain elevated access to the machine, and then installs rootkits to hide his traces from the system administrator and anti-malware utilities. A rootkit's function is to hide all traces of malware activity on the machine, which includes malware processes, network connections used by malware, files used, and registry entries used by malware from system administrator 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 will not survive a system reboot, as they operate only on system memory and do not modify any files on disk. But persistent rootkits change the persistent files on disk to load themselves on a system reboot. It is comparatively easier for anti-rootkit tools to catch persistent rootkits by checking the integrity of critical disk data before shutting down the machine. Rootkits are gaining more attention these days and are becoming serious security threats because of the emerging sophisticated attacks.

In the next few sections, we classify different types of rootkits, provide an overview of existing techniques to detect rootkits, and discuss why they are not sufficient to tackle the emerging sophisticated rootkits.

2.1 Common Exploit Techniques by Rootkits

Rootkits modify the execution flow of the OS to hide malware activities from the system administrator. Rootkits can operate both in user space and kernel space, depending on the exploitation level.*Kernel mode rootkits* are more detrimental than *user mode rootkits* because of the unrestricted access privilege. They can manipulate any software component via the compromised OS and hence are very hard to detect. Rootkits use the following techniques to achieve malware's stealth and subvert the system.

2.1.1 Import Address Table Hooks (IAT):

An IAT hook is a technique that was commonly used by naive rootkits at the user level. Import Address Table (IAT) contains function pointers that lead to the functions in different shared libraries. When a user-level process makes a function call that is implemented in a shared library, the IAT is traversed to get the address of that particular function before transferring the control. User-level rootkits patch this table to hook these function pointers and install trampoline functions to filter data.

2.1.2 System Service Descriptor Table Hooks (SSDT):

SSDT, also called a system call table, is in the kernel space, which contains function pointers to handle different system calls. A kernel mode rootkit can modify these SSDT entries and replace a function pointer with an address of its own to hijack the system. Loadable kernel modules (LKMs) have access to SSDT, and hooking SSDT is a simple and popular attack accomplished by LKMs. To know the system state, all the system administrator utilities use system calls, and it is easy to intercept these calls and filter data.

2.1.3 Interrupt Descriptor Table Hooks (IDT):

IDT is another table in the kernel space used to store the interrupt handlers in the kernel. The kernel mode rootkit can modify an entry in IDT to replace the legitimate interrupt handler with the fake handler. Keylogging malware uses this technique to intercept keystrokes of interest, e.g., passwords, social security numbers, bank accounts, without any knowledge of the user.

2.1.4 Direct Kernel Object Modification (DKOM):

In the DKOM technique, the rootkit modifies the OS data objects directly to remove the information pertaining to the processes the malware intends to hide. For example, the rootkit can modify the linked list that the "ps" command uses to find what processes are running. It removes the node of the linked list that has information about the malware process and hence the utility tools only see this manipulated linked list and will not report any unintended use of computing resources. This technique is hard to detect- because it is very difficult to track changes in the OS data.

2.2 Sophisticated Rootkits

Virtual memory subversion is a technique used by the Shadow Walker Rootkit [34] in which the memory contents are faked when integrity-checking tools read the pages occupied by malware. Typically, integrity-checking tools scan physical pages to watch for any modification by malware. This rootkit tries to hide malware's activities by returning the original legitimate data when integrity-checking tools try to read these pages. To accomplish this, the TLB is flushed for malware pages so that any memory access to these malware pages walks through the page tables. The compromised OS then manipulates the page table entries of these malware pages and invalidates the PTEs so that there will be a page fault for every access to these malware pages. Then, the patched page fault handler will differentiate between memory read and execute

operations, return the original legitimate data if it is a non-execute memory access or the modified contents if it is a memory-execute operation for malware to execute. This makes all integrity-checking tools useless.

Subvirt is another complex rootkit that was recently demonstrated [16]. This rootkit makes the host OS a virtual machine and installs a virtual machine monitor below the host OS. To accomplish this, the boot files are modified so that when the system reboots, VMM boots before the host OS starts and the VMM takes full control of the host OS. The host OS will not even know that it is executing as a virtual machine on top of a VMM. Also, this virtual machine-based rootkit installs other guest malware OSes completely isolated from the original host OS. *Bluepill* is another conceptual rootkit [29] that makes use of advanced hardware-assisted virtualization support to take control of the host OS without changing any system files. Using secure virtual machine (SVM) in AMD-V technology and Intel's VT-X technology, Bluepill installs a thin hypervisor below the host OS on-the-fly and downgrades the host OS to become a virtual machine without modifying the boot files. It is proved that, today, we cannot effectively prove or detect virtualization-based rootkits [25]. Cloaker [6] is a recent rootkit that exploits hardware to conceal itself without modifying the OS code and data. One of the hardware configuration registers is modified to change the location of the interrupt service routines (ISRs) and install malware in this virtualized environment without modifying the host OS image. Recognizing the sophistication of these emerging rootkits, we propose a solution at micro-architectural level to control all software layers above the bare hardware. This solution should be effective in identifying hidden software contexts in any software layer, including the VMM or hypervisor level.

2.3 Software Anti-Rootkit Techniques

The existing software anti-rootkit techniques use one of the following methods to examine the corrupted system and trigger an alarm:

2.3.1 Signature-based detection:

In this detection scheme, the memory is scanned to find the sequence of bytes that comprise the fingerprint of known rootkits. If there is a match with known fingerprints, an alarm is triggered. The downside of this approach is that it can be used to detect only known rootkits with known fingerprints.

2.3.2 Heuristic/Behavioral detection:

In this scheme, a deviation of the expected normal system behavior is used as a clue to detect potential suspicious activity. For example, using the execution time as one heuristic, if the execution time of a system call has consistently increased, we can infer that there is additional code inserted by malware and trigger an alarm. The downside of this approach is that it triggers many false-positives because there can be deviations in these heuristics that are not deterministic.

2.3.3 Cross-View-based detection:

In cross-view-based detection, a high-level system view obtained by high- level OS functions is compared with a low-level system view obtained by very low level OS data. Any mismatch triggers an alarm and concludes that the high -level OS view is changed because of manipulation in intermediate OS layers. Rootkit Revealer [24], Klister [17], Blacklight [3], and StriderGhostbuster [11] use this technique.This detection scheme assumes that the low-level OS view cannot be modified by the rootkit and it cannot get very complex. But today, rootkits are very complex and this detection scheme will also fail.

2.3.4 Integrity-based detection:

In integrity-based schemes, a current snapshot of system memory is compared with a trusted baseline. Any mismatch is taken as evidence of suspicious activity. Tripwire and System Virginity Verifier [27] were developed based on this technique.

The approach followed by current software techniques is inherently flawed because they operate in the same corrupted software stack. This results in a endless battle between rootkit and anti-rootkit camps. Subvirt, Shadow Walker, Bluepill, and Cloaker are new rootkits that are very sophisticated and indicate the complexity of future rootkits.

2.4 Hardware Anti-Rootkit Techniques

The Copilot hardware detection scheme [23] followed the right approach of having an OS-independent hardware solution to check the integrity of the host OS 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 integrity is continuously checked. A counter attack against this system was demonstrated by Rutkowska [30] in which it creates different views of the system memory to the processor and PCI device to subvert CoPilot solution.

CHAPTER III

EXPLORING ARCHITECTURAL SOLUTIONS

As discussed earlier, none of the software solutions are strong enough to defend the system against rootkits. Detecting hidden VMMs is claimed to be impossible using software techniques. This makes it necessary to have an OS -independent hardware solution that enhances the relationship of hardware and OS and makes the system more security aware. As stealth is the most common exploitation of rootkits, we focus on the stealth execution of software contexts achieved by memory-based kernel rootkits in this work. Viewing the problem of stealth from the hardware perspective, if the hardware has the capability to identify process contexts, the hardware can expose the list of software contexts making use of hardware resources to the system administrator. The system administrator can view this list as the master list of processes that cannot be manipulated and compare it with the software returned list of processes. Any mismatch implies that the software stack is trying to hide the execution of a few processes and it can trigger and alarm. Before proposing a new micro-architectural solution, we explored the current architectures to determine whether small changes to the current architectures will provide hardware the capability of recognizing software contexts. In the following sections we discuss the possible solutions and their weaknesses.

3.1 Tagged TLB

First we considered using tagged TLB and using the contents of the PID register to identify processes. In fact, many processors use tagged TLB to avoid flushing TLB on every context switch. This requires the OS scheduler to load the PID of the upcoming process in the PID register, which will be compared with the PID stored

in the tagged TLB when there is a memory access. Using the PID register, a secured hardware list of processes can be exported with the PIDs of all running processes. In the beginning this appeared to be a good approach to track every software process making use of hardware resources. But when the security evaluation was done, we saw that this does not prevent the OS from manipulating the hardware list of PIDs. The compromised OS can use the legitimate process' PID to run malware. As the TLB is tagged, it might contain VPN-PPN mappings of the legitimate process, which will be returned when the malware process is run. To take care of these incorrect address mappings, the TLB has to be flushed by the OS before context switching to the malware process. For example, in the x86 architecture, the TLB can be flushed using the INVLPG <address> instruction.

3.2 Tracking based on PDBA

In x86 architectures, every running process has a unique page global directory (PGD), and upon a context switch to make the context switching simple, the OS has to load the page table base address (address of the PGD) into the CR3 register. The CR3 register is used to do a hardware page table walk when there is a TLB miss. Knowing that the CR3 value of every process is unique, we thought we could use this to identify every process making use of hardware resources. In the beginning this looked like a good approach because the compromised OS cannot run the malware process by making use of the PGD base address of the legitimate process. If it is used, the malware process will fail to see its address space and will fail to execute. The dependency between CR3 and the execution context of the malware process appears to make the security scheme strong. Nevertheless, it is easy for the compromised OS to subvert this. It is as simple as swapping the page tables of a legitimate process and malware process before context switching to the malware process and using the PGD base address of the legitimate process. This will not expose the PGD base address of

the malware process to hardware.

CHAPTER IV

SHARK: 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 the OS's direct intervention will always fail. This makes it necessary to design a new processor architecture that has the capability to recognize process contexts. The rationale behind the process context aware architecture includes (1) using hardware support for creating a new process, (2) isolating and protecting process' address space in a hardware-hardened sandbox so that it cannot be circumvented by the compromised OS, (3) providing the capability to recognize the running process without any direct intervention of the OS. The hardware will be able to identify the running process without any dependence on the vulnerable software stack. This enhances the relationship between the hardware and OS and makes it possible to achieve a process context-aware architecture.

To achieve the above set of objectives, we propose a process context-aware architecture called SHARK, which stands for- Secure Hardware Against RootKits. In the proposed scheme, the master control of processes is delegated to the SHARK Security Manager (SSM), which is a hardware engine for enforcing the security of different process contexts. The OS carries out its regular operations under the supervision and assistance of the SSM. Figure 1 gives an overview of the proposed architecture, including the software mechanisms. The rootkit detection capability

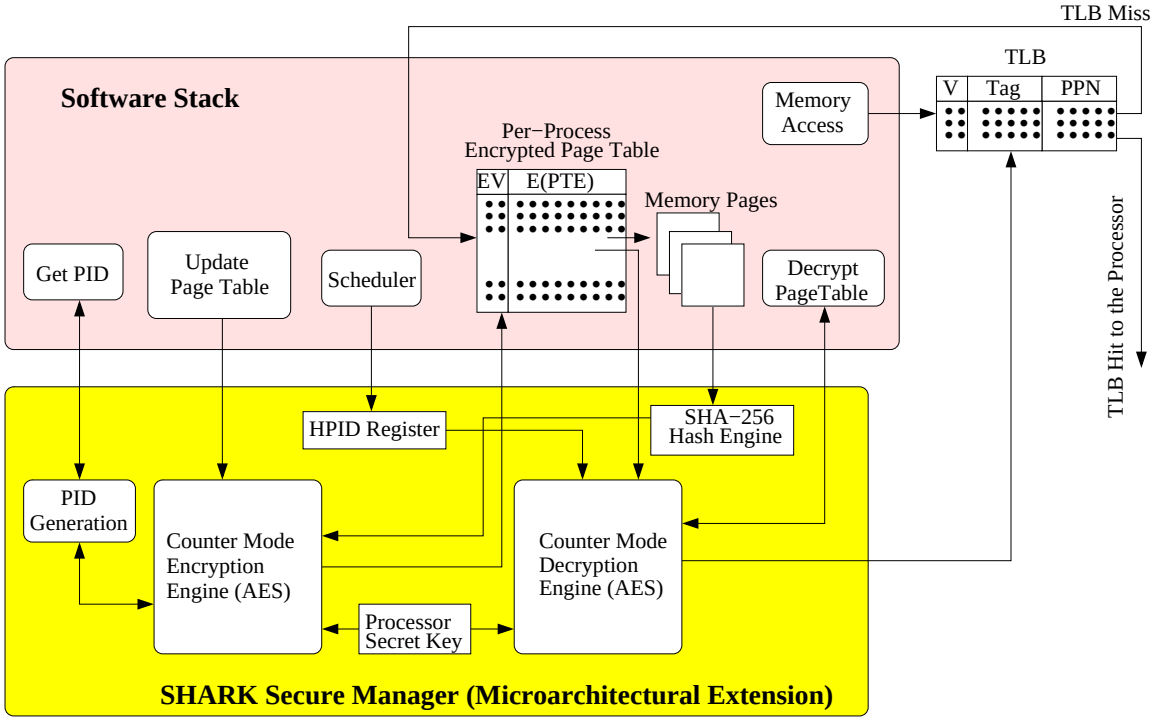


Figure 1: Architectural support for SHARK processor.

is achieved by integrating the following components into one processor: *Hardware-Assisted PID Generation,Process Page Table Encryption and Decryption, and Process Authentication.* These components are implemented within the SSM, a hardware-based micro-architectural extension that works seamlessly with the software stack. More details about each component are explained in the following sections.

4.1 Hardware-Assisted PID Generation

We have already seen that the PIDs generated by the OS are vulnerable to attacks, and using this conventional approach, we cannot prevent the OS from using different PIDs to run and conceal malware. So, *Process ID Registration* is the first attribute that we introduce in the SHARK architecture. This registration of every new process should be done by the operating system before the hardware gives the respective process, the permission to make use of hardware resources. In other words, when a new process is created, the Shark Security Manager (SSM) generates a new PID

for the process and not the vulnerable OS. Note that this hardware- generated PID need not be a secret, as it is simply used as a counter value in our counter-mode encryption [7], to be described later. When the SSM gets a request from the OS to generate a PID, it generates a 64-bit PID by just incrementing the PID counter and returns the same to the OS. Even if a new process is created every cycle on a 1GHz processor, it takes 584 years for the PID counter to overflow which is long enough for the system to reboot and initialize the counter. This makes the hardware PID generator very simple without any PID pool management logic. Thereafter, the OS has to use the same PID whenever it has to run the respective process. On a context switch, the OS has to load the PID of the upcoming process into the HPID register which is an integral part of the SSM.

4.2 Process Page Table Encryption and Decryption

Generating the PID of every process in hardware and asking the OS to load the PID of the respective process into the HPID register on every context switch will not prevent the OS from using the PIDs of legitimate processes to run and conceal malware processes. This makes it necessary to establish an enforced dependency between the PID of the process and the execution of the process itself. This dependency can be achieved by the the proposed *Process Page Table Encryption/Decryption* using the PID of the respective process as the counter used for counter mode encryption. If the compromised OS tries to use a different PID to run a particular process, it breaks the dependency and will prevent the process from running. Before getting into the details of page table encryption, we briefly review the counter-mode encryption scheme.

4.2.1 Counter Mode Encryption

Counter-mode encryption is a common symmetric-key encryption scheme [7]. It uses a block cipher (e.g., AES [8]), a keyed invertible transform, that can be applied to short fixed-length bit strings. To encrypt with the counter mode, one starts with a

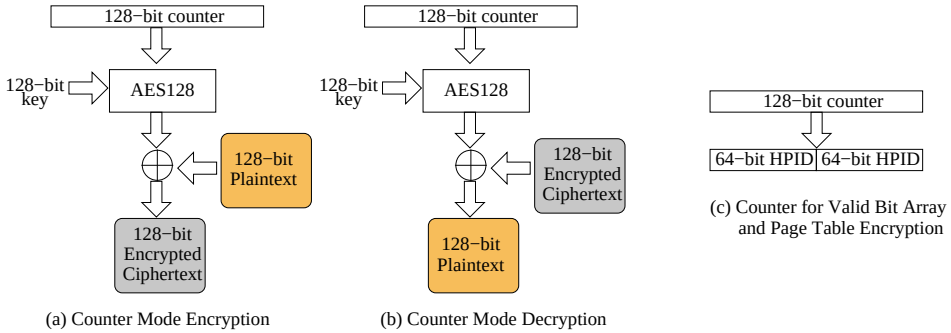


Figure 2: Counter-mode encryption and decryption

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 ciphertext, 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 [2]. 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 the OS creates a new process, it requests the SSM to generate a new PID for the respective process and provides the address of the page global directory (PGD) and also the first page table entry of the respective process. The SSM generates the PID as mentioned in section 4.1. Using this newly generated PID as a counter for encryption, the valid bit array of the PGD and the PTE mapping in the last level page table are encrypted before returning the generated PID to the OS. In this section, we

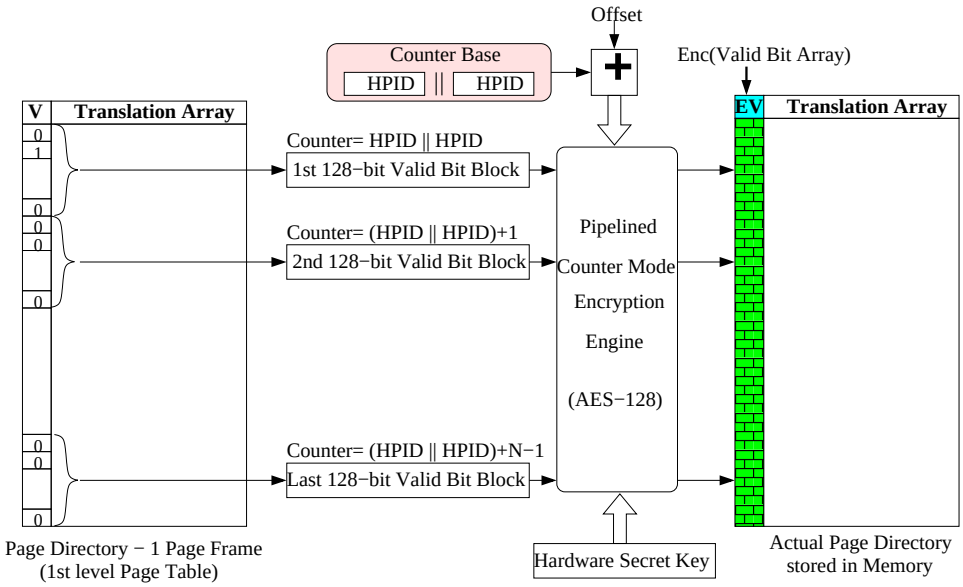


Figure 3: Valid bit array encryption

describe the valid-bit array encryption in the PGD. In section 4.2.3, we describe the details of PTE encryption.

A hardware secret key is first implemented that cannot be read out by any means similar to what was used in prior secure processors [19, 33, 35]. Using the hardware-generated PID and the secret hardware key, the first step is to encrypt the entire valid-bit array of the PGD in blocks of 128 bits (i.e., every 128 entries) using the counter- mode encryption scheme. The 128-bit counter value for encryption is formed by concatenating the hardware-generated 64-bit PID value, as shown in Figure 2(c). This guarantees that every process has a different valid-bit array. Starting from 0 for the first 128-bit block, the counter value will then be incremented by one for each encrypted block.

Figure 3 shows the entire encryption process of the valid-bit array. The result of this encryption is an encrypted bitstream stored in the original valid-bit array of the page table. After this initial encryption, the PID that was used as the counter for this encryption will be returned to the OS. Note that this PID is returned only after the initial encryption. The overhead of encrypting the valid-bit array of the PGD is

1024 V-bits/128 (input block size for AES) * 80ns = 640 ns without using a pipelined AES implementation, which is negligible compared to the lifetime of a process.Note that the 80 ns overhead for AES encryption is much slower than what is reported in more recent AES engine implementation fabricated using a 0.18μm process in [12].

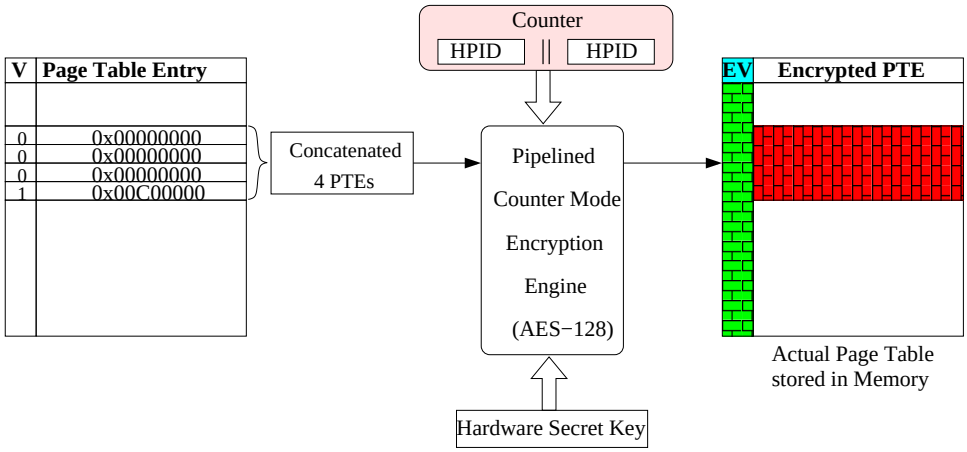
By means of this initial valid-bit array encryption, we establish the dependency between the SSM and the software stack using the HPID and the hardware key. For any subsequent page table walks to be valid, the OS cannot misguide the hardware by using a fake PID, as the decryptions of the already encrypted page tables will fail.The importance of valid-bit array encryption will be explained again in section 4.7.

4.2.3 Page Table Translation Encryption and Updates

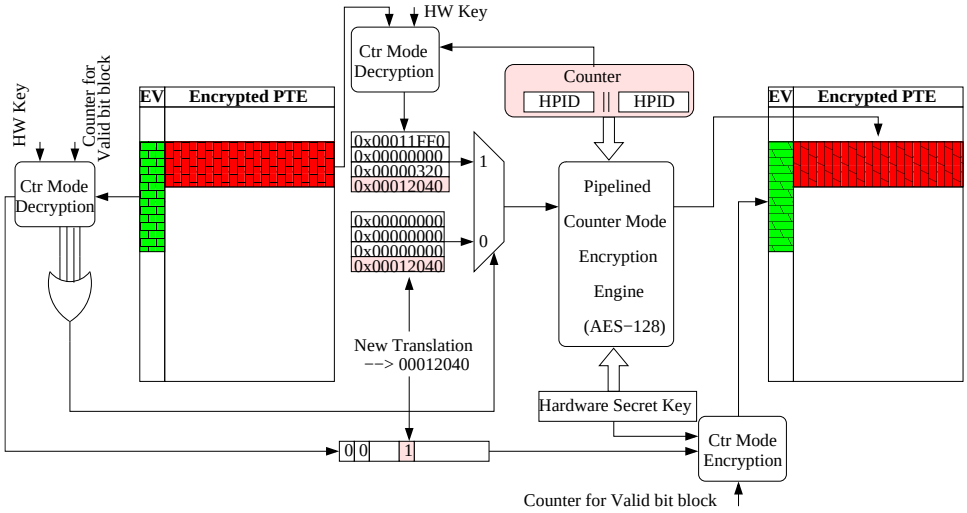
The translation of the last level page table (PTE) is also encrypted. This section gives details about this encryption. Again, for this encryption, we use counter mode encryption and use a counter value, as shown in Figure 2(c). There is no extra memory overhead or extra lookup logic involved for storing and searching the counter values. These counter values are nothing but the PIDs of the respective processes stored in the process context information.

The encryption granularity depends on the maximum physical memory that can be supported in an architecture. For example, the x86-64 architecture supports a maximum physical address of 40 bits in the IA-32e mode [14]. Given a 4KB page, the physical address excluding the offset is 28 bits. According to the AES standard, the block size for encryption and decryption is 128 bits. Hence, we propose to encrypt four consecutive PTEs at once. Note that the current Linux uses a 4-byte integer for each PTE and hence four consecutive PTEs are enough to store the encrypted text. Therefore there is no memory overhead introduced in an actual implementation.Figure 4(a) shows the encryption process.

When the OS needs to update the process page table, it has to request the Shark



(a) Encryption Process



(b) Decryption Process

Figure 4: Page table update in SHARK.

Security manager to do this. Since the v-bit array of the PGD is encrypted, the SSM first decrypts the v-bit array of the PGD to continue with the page table walk. Then, it reaches the last level page table and examines the four PTEs that contain the target translation. It first decrypts the corresponding 128-bit v-bit array block of the last level and if none of the four PTEs is valid, SSM simply sets the valid bit, re-encrypts the 128-bit valid-bit block, and encrypts the new translation with the neighboring invalid mappings. However, if any of the other three is valid, which means that the 128-bit encrypted PTE contains some valid mappings, SSM decrypts the encrypted PTEs, adds the new translation, and re-encrypts to update the page table correctly. Figure 4(b) details this procedure. To maintain the correctness of the contents of the page tables, every update to the page table should go through the hardware-based SSM.

4.3 SSM-managed TLB updates

If there is a TLB miss, the TLB has to be refilled by a hardware page table walk. We target the widely distributed machines used such as the x86 and PowerPC, that use hardware-managed TLBs and we build our security module on top of it. Since the TLB cannot be tampered with in our SHARK architecture, we store the plaintext translation like a regular TLB. Since the page tables are encrypted in memory, every TLB miss is followed by a hardware page table walk and a series of decryptions before refilling the TLB. Using the PID value of the respective process from its process context, the SMM first decrypts the corresponding 128-bit valid-bit block in the page directory. Then it does a hardware page table walk to reach the last level page table, decrypts the corresponding 128-bit valid-bit array block, and finally decrypts the encrypted PTEs before refilling the TLB. Since the right PID value should be used for correct decryption of page tables, it is compulsory for the compromised OS to load the PID of the malware's process and it has to reveal its PID to the hardware

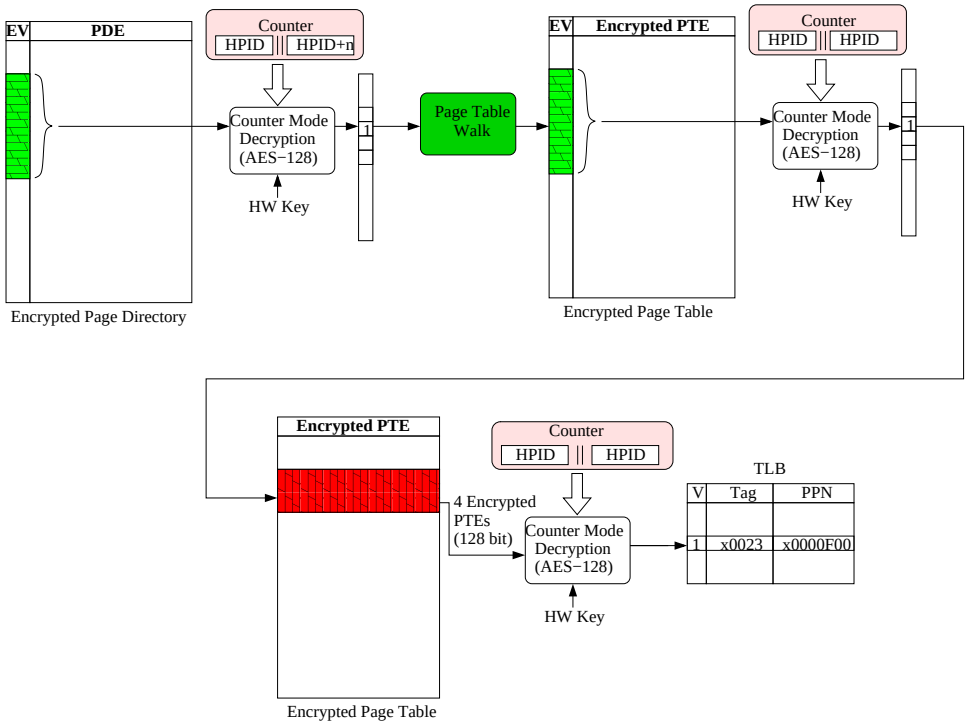


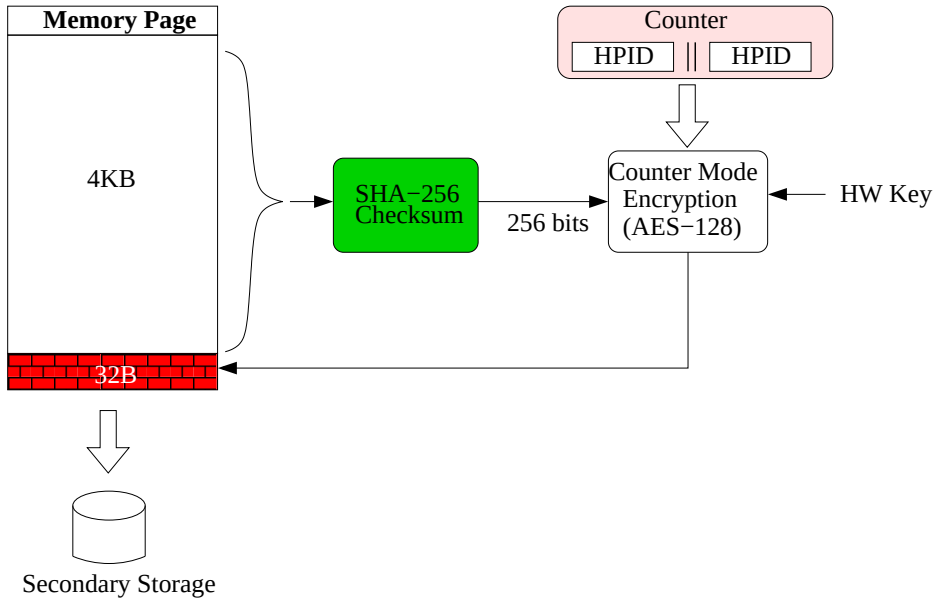
Figure 5: TLB update handled by the SSM.

whenever it executes. This is depicted in Figure 5.

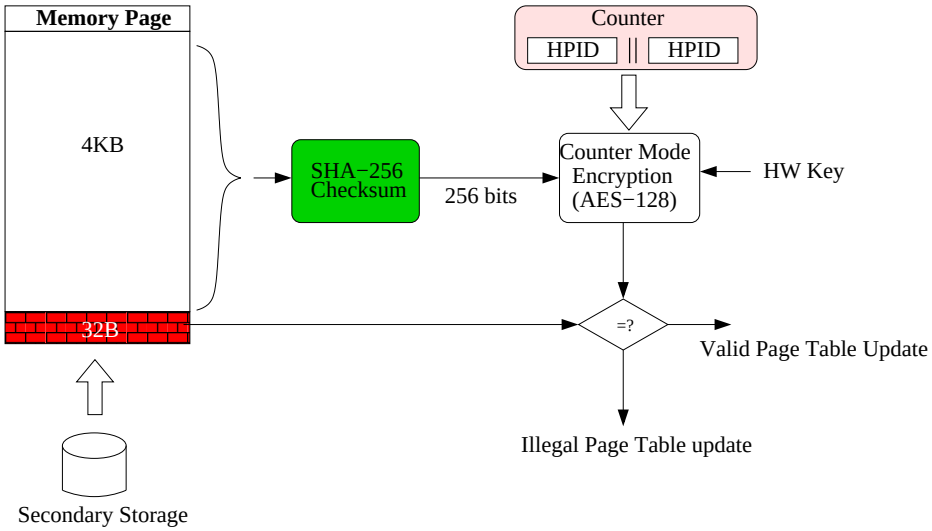
4.4 Instructions supported in SHARK

Three new instructions are supported in the SHARK architecture, which are listed in Table 1. The GENPID instruction has to be used by the OS when the first memory page is assigned to a newly created process' address space. As described in section 4.1, when there is a request from the OS to generate a new PID, the SSM increments the PID counter and uses the same PID as the counter to encrypt the following: (1) valid-bit array of the page directory, (2) valid-bit array of the last level page table, (3) page table translation (VPN-PPN) in the last level page table, It then returns the generated PID back to the OS. The MODPT instruction is required by the memory management module of the OS to directly update the page tables of processes whenever the kernel swaps out and swaps in memory pages. MODPT decrypts the encrypted PTE, inserts the new mapping and re-encrypts the PTE back to maintain

correctness. In addition to this, in order to track the physical pages of this particular process, if we are invalidating the PTE (e.g., when swapping a page out), the MODPT needs to compute the SHA-256 [32] checksum of the memory page and encrypt the checksum using the PID of the process as the counter for encryption before swapping out the page.This is illustrated in Figure 6(a). The resulting 32B encrypted checksum accounts for 0.8% space overhead for every 4KB page, to prevent a particular attack, as described in section 4.7. This extra operation is designed to further strengthen the association of the memory page and the owning process.The reverse operation is performed when a MODPT is used to validate a previously mapped PTE (swapping a page in). Before updating the respective PTE, it checks whether the memory page is owned by the process.(Note that this check is not performed while swapping-in the memory page for the first time and the MAPPED bit (M-bit) in every PTE keeps track of whether or not the virtual page is mapped 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 today's Linux kernel implementation.) This authentication is achieved by computing the checksum of the memory page, encrypting the checksum, and comparing it with the stored encrypted checksum. Match in the checksums implies that the memory page is truly owned by the process and the update is legal. After this authentication, the PTE of the page table is modified, which is illustrated in Figure 6(b). Note that because of maintaining and encrypting the checksum of the page when we are moving the physical pages from one PTE to another, SHARK always keeps track of the ownership of this page and so the compromised OS cannot modify the PTE of a legitimate process and map it to a malware's page to conceal malware's page tables. Finally, the DECPT instruction is required if the kernel wants to directly read the page table contents.



(a) Swapping Out Memory Pages



(b) Swapping In Memory Pages

Figure 6: Security enhancement for using the MODPT instruction

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 (swapping-out), SHA-256 hash of the memory page is computed and encrypted before swapping out the page (c) If M-bit = 1 and MODPT is used to validate a memory page (swapping-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

4.5 Process Authentication

If the compromised OS tries to use a hijacked PID from a different process, the end result of the process page table decryption will result in an incorrect physical page number, which, if used, will prevent the execution of malware. 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*.

4.6 Stealth Checker

This section discusses the last component of SHARK which is called *Stealth Checker*. With the SSM hardware extensions, every running process is controlled and revealed by the hardware. These hardware extensions make sure that the OS cannot fool the hardware and enforces the OS to reveal the details of every running process context to the hardware. But these hardware extensions cannot prevent OS from manipulating software system administrator utilities to hide malware processes. For example, even if the hardware knows about every running process, it does not prevent the malware from manipulating "ps" and "top" to hide malware processes from system administrators. So the hardware list of processes is meaningless if it is not safely exported to the system administrator. This exported master list of processes can be compared 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 the above mentioned scheme to be secure and effective, we have to prevent the compromised OS from intervening and subverting the master list of PIDs exported by the hardware. Designing such a trusted feedback passage between the hardware and anti-rootkit software is very crucial for identifying stealthy processes.

We propose to implement the stealth checker in firmware which is caller prior to every context switch. Every write to the HPID register triggers this exception and the exception handler reads the contents of the HPID register and gets the PID if the upcoming process. There are no security implications till 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. This small

packet of 128-Bytes accommodates 64-bit PIDs of 16 processes. The firmware sends this data once every 10 context switches, which reduces the network activity even more. Unlike CoPilot technique, which send out plain memory pages for examination, our firmware encrypts this data using a 128-bit key assigned by the system administrator on firmware installation. Even though we are sending the data packet to the remote machine by making use of insecure OS services, the OS cannot compromise this list as it is encrypted. To make it more secure, sequence numbers can be employed in each data packet to prevent OS from using replay or blocking attacks. If the OS attempts to block the packets sent by the handler, the system administrator sitting on the remote machine can conclude that the OS is compromised and take appropriate actions. On the other side, the remote machine can decrypt the data packets from the host machine and maintain an event log of process contexts running on the host. 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. This reduces the overhead of Stealth Checker below 0.2% and makes it highly practical.

4.7 Strength of SHARK

In this section, we discuss the potential future threat models and we analyze how SHARK processor can defend against these attacks. We thought about many future exploits that can be devised to subvert SHARK knowing that the OS cannot be trusted. SHARK has the capability to prevent all these malicious attempts carried out by untrusted OS.

First, if rootkits hijack a legitimate process' PID to conceal malware's PID, it results in incorrect decryption of address mappings and makes sure that the malware hiding process does not execute. Note that the encryption is seamless, established using the HPID when the process and its page directory are created.

In another attack, the rootkit may plan to encrypt the page tables of a malware process using the PID of a hijacked process. Once this is achieved, the malware process can always run by using the PID of the legitimate process. This attack will fail because any update to the page table has to first decrypt the Valid-bit array of the PGD. If the rootkit tries to use a different PID in between and update the respective page table, the valid bit array of PGD will result in incorrect decryption.

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.

In another attack model is to have the malware invalidate all the allocated malware pages and swap all the malware 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, dues to the encryption of the valid bit array of the last level page tables. 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 OS cannot simply update the page table with its own encrypted valid bit array and mappings since it knows both the PIDs and the page tables are in memory. This is impossible since the hardware burn-in secret key cannot be read by the OS by any means. It is hardwired into AES engine for performing encryption and decryption. This makes this threat model not feasible.

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 4.4 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 use other process' PID to use its memory pages while it is still executing.

Last but not least, we know that sophisticated virtual machine-based rootkits [16, 29, 25] are emerging these days; we will discuss the implications of SHARK architecture on these rootkits in this section. In virtual machine-based rootkits, the malicious software uses either hardware support for virtualization or modify the boot files so that the VMM boots under the host OS. Once the VMM starts operating under the host OS, it is completely compromised. Now let us discuss the challenge of identifying the nested VMM's installed using hardware virtualization support in [25]. 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 revealed to system administrator.

CHAPTER V

EXPERIMENTAL ANALYSIS

Two sets of experiments were conducted to evaluate the proposed SHARK architecture. 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, several rootkits were installed on Linux OS running on top of an emulated SHARK architecture. To emulate the entire system, including the SHARK security manager, Bochs, a highly portable open source x86 PC emulator was used.

The proposed scheme was verified to be practical by modifying the memory management unit, process management unit and the scheduler of the Linux kernel versions 2.2.14 and 2.6.16.33 to use the SSM implemented in Bochs to support our proposed mechanism. Using these new instructions supported by SHARK (shown in Table 1), the kernel was modified. 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 access and use appropriate counter value for decryption. Shared libraries is not an issue because of if two page tables lead you to a shared physical page, different PIDs will be used to encrypt their respective page tables. Also the SHA-256 hash for the shared page can be encrypted using the respective PIDs of the processes, sharing the memory page. In virtualization systems, the lowest layer, i.e. the hypervisor,

must use the ISA support provided by SHARK.

The security evaluation was performed by installing rootkits over the modified kernel for SHARK running over the emulated SHARK hardware architecture. The following five rootkits collected from [22] 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 the above rootkits are for different kernel versions. The first three rootkits attack Linux 2.2 kernel, while the last two were developed for the linux 2.6 kernel. These rootkits were inserted as LKMs and can access the kernel space and can 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`). Using this setup we contrived a compromised software stack to be able to assess the effectiveness of our SHARK architecture. The base kernel's scheduler was modified to load the HPID register with the PID of the process prior to each context switch and every write to the HPID triggered an exception service by the stealth checker. As described in section 4.6, the exception handler in the firmware cannot be compromised as it is a firmware. 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. The rootkit tries to hide the information of malware processes from this list. By comparing these two lists, hidden malware processes were detected and it triggered security alarms to reveal the processes running in stealth, demonstrating effectiveness of our SHARK architecture.

5.2 Performance Evaluation

In order to protect process page tables, SHARK introduces extra encryption/decryption overhead. In this section we evaluate this overhead's impact on performance. Cycle information was obtained from Virtutech's Simics [20] with its gcache model enabled.

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	80	138	200
Config2			8-way, 64B line	160		
Config3			4MB, 19 cycles	80		
Config4			16-way, 64B line	160		
Config5	4GHz	32KB, 8-way 64B line, 3 cycles	2MB, 25 cycles	160	276	300
Config6			8-way, 64B line	240		
Config7			4MB, 38 cycles	160		
Config8			16-way, 64B line	240		

Note that we could not perform functional evaluation using Simics because some modules e.g., Page Table Walks, are not open sourced and hence we could not modify them to model our SHARK implementation. 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. 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 [18] 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 [15]. 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 [5]. Then we increase

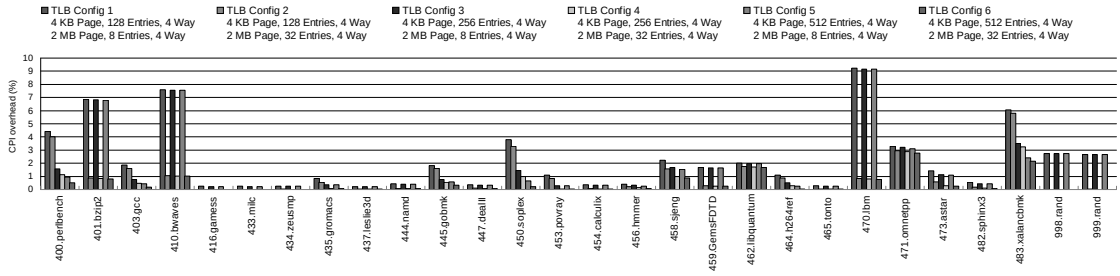


Figure 7: Performance impact with different TLB organizations (Config1)

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.

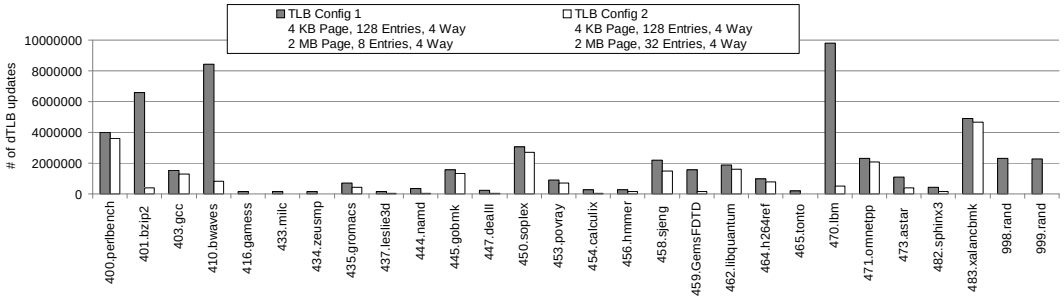


Figure 8: Number of D-TLB updates for TLB Config1 and TLB Config2

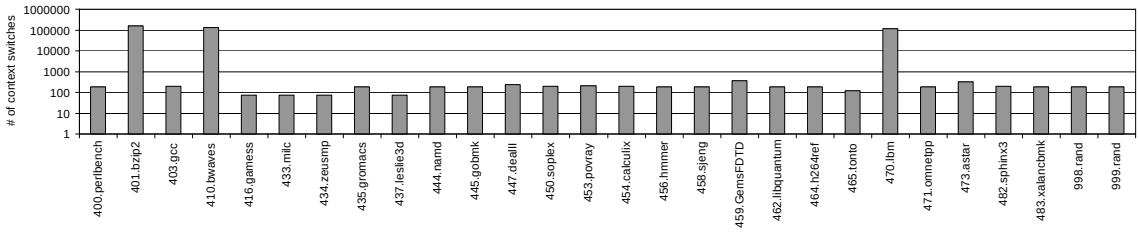


Figure 9: Number of context switches (amid 2 billion instructions)

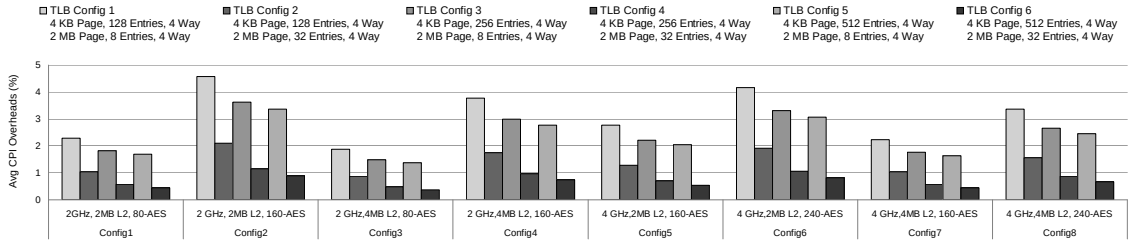


Figure 10: Average overheads for all the benchmarks with different configurations

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

CHAPTER VI

RELATED WORK

To detect kernel mode rootkits, many software techniques have been proposed [4, 26, 36, 28, 27]. These software solutions operate in the same corrupted software stack and they expect that some kernel components cannot be compromised. But, sophisticated rootkits subvert these trusted kernel components to defeat anti-rootkit solutions. These software solutions were dependable when they were released, but because of the increasing complexity of the rootkits, they are not considered to be secure. The hardware solution, CoPilot, mentioned in section 2.4, was proven to be insecure by Joanna Rutkowska in [30]. Also, note that this is not a micro-architectural approach, but rather a system-level solution that was proposed to deal with the problem.

Many researchers have proposed the idea of checking the integrity of the host OS using virtual machine monitors (VMMs) [9, 31]. These VMMs are typically optimized to be a thin software layer, and the security manager inside the VMM verifies the integrity of the host OS. These techniques are no longer safe because of rootkits, like Blue Pill , that are exploiting hardware virtualization support [29]. Bluepill is proved to work under the XEN hypervisor [37]. So we cannot trust the VMM and run anti-rootkit tools in VMM too. It is shown in [25] that, today, none of the techniques can detect virtualization-based rootkits. SHARK is a micro-architectural solution and that can address virtual machine-based rootkits effectively, as discussed in section 4.7.

Untrusted OS is not a new problem for the micro-architectural research community [35, 19]. 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 possible 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 rather 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. We cannot use Intel's TPM technology [13] to solve our particular problem. Even though TPM provides a trusted base that cannot be modified by the corrupted OS, we cannot use this feature to recognize processes in TPM firmware. If TPM has to control processes running in OS, it has to run below the OS as a VMM. It is not possible to implement the VMM in TPM as TPM runs in its private memory and hypervisor is meant to run in shared memory under the host OS. To the best of our knowledge, we are the first to propose micro-architectural support, to enhance the security of the OS to deal with applications running in stealth.

CHAPTER VII

FUTURE WORK

In this section we describe the possible extensions of SHARK that will be very useful for similar security applications.

We know that SHARK just operates at the process context level and all the threads of the process share the same PID. To have a more fine-grained control of software contexts, the next step would be to authenticate every thread running on the system. This would help if the malware were just spawning new threads and running as a part of a legitimate process.

The other possible extension to SHARK would be to associate every network connection (I/O) with its owning process and expose this to the system administrator. This would be useful if the network were exploited by some hiding malware in the system. It would provide an opportunity for the system administrator to know what exactly is happening on the bare hardware.

Virtualization has become the modern trend and is widely used to abstract software stacks away from underlying hardware resources. Since the lowest layer of software (VMM) has the entire responsibility of securing the system and because of emerging attacks on VMMs to subvert the enitre software stack (e.g., widely used XEN hypervisor is proved to be vulnerable), we have to closely inspect the loop holes of VMMs, which can be exploited by hackers. It would be very beneficial to protect the integrity of this lowest-layer of software(hypervisor) by securely sand-boxing this layer of software. Instead of running the hypervisor in the shared memory that is accessible to all the guest domains, we propose running this critical software layer in a protected memory region that is accessible to a single master core. This results in

complete physical isolation of the hypervisor memory from guest OSes. This master-core also has access to the shared memory of the guest-cores and hence can provide hypercall services to guest-cores. Master-core does not run any guest domains and is available only for hypervisor services, making it impossible for guest domains to modify the hypervisor memory. Additional hardware support should be provided for hypercall communication from slave cores to the master cores.

CHAPTER VIII

CONCLUSION

Rootkit-based exploits have become a serious concern in cyber-security. 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 thesis, proposes an autonomic architecture called SHARK that operates against stealth achieved by rootkits' exploits. To the best of our knowledge, this is the first work 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.

REFERENCES

[1] ARM, "ARM TrustZone Technology."

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

[3] Blacklight, "http://www.f-secure.com/blacklight."

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

[5] Dadda, L., Macchetti, M., and Owen, J., "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.

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

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

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

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

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

[11] GhostBuster, "http://research.microsoft.com/Rootkit/."

[12] Hwang, D. D., Tiri, K., Hodjat, A., Lai, B.-C., Yang, S., Schaumont, P., and Verbauwhede, I., "AES-Based Security Coprocessor IC in 0.18μm CMOS with Resistance to Differential Power Analysis Side-Channel Attacks," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 4, pp. 781–791, 2006.

[13] Intel, "Intel Trusted Platform Module Technology."

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

[15] Kgil, T., Falk, L., and Mudge, T., "Chiplock: support for secure microarchitectures," *SIGARCH Computer Archititecture News*, vol. 33, no. 1, pp. 134–143, 2005.

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

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

[18] Labs, H., "CACTI 4.2."

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

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

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

[22] PACKETSTORM, "http://packetstormsecurity.org/."

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

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

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

[26] Rutkowska, J., "Detecting Windows Server Compromises with Patchfinder 2," in *www.invisiblethings.org/papers/rootkits _detection_with_patchfinder2.pdf*, 2004.

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

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

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

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

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

[32] SHA-256, "National institute of science and technology fips pub 180-2: Sha256 hashing algorithm,"

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

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

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

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

[37] Wojtczuk, R., "Subverting the Xen hypervisor." http://www.blackhat.com/presentations/bh-usa-08/Wojtczuk/BH_US_08_Wojtczuk_Subverting_the_Xen_Hypervisor.pdf.