SHARK: Autonomic Protection Against Stealth by Rootkit Exploits

Abstract

This paper proposes an autonomic architecture called SHARK that operates against stealth achieved by malicious Rootkit's exploits. SHARK employs secure hardware support to provide system-level security, without trusting the software stack including the OS kernel which is assumed un-trusted. These kind of attacks are typical to Rootkit exploits that make use of kernel's vulnerabilities to get root privileges and continue to run their malware applications on compromised machines. These malware processes operate completely in stealth, leaving no trace to the system administrators. Seeing the need for hardware support in the architecture to capture such malware operating in stealth, SHARK is proposed to enhance the relationship between the OS and the hardware architecture, making the entire system more security-aware.

The proposed process context aware architecture gives hardware knowledge about software contexts running over it. This helps system administrators to obtain feedback directly from the hardware to reveal malicious processes in stealth, even when the OS kernel is compromised. As indicated in our experimental results, this is highly effective in identifying rootkits that employ different schemes in software to hide malware processes. In addition, the performance analysis shows that the overhead of the SHARK mechanism is very minimal, making it highly practical.

1 Introduction

The security of a computing system highly depends on the vulnerability of its underlying operating system. As the complexity of a modern OS increases, they are becoming more and more vulnerable to attacks that directly compromise the security of the whole system. To warrant a highly secure computing system, the kernel security is becoming very crucial. As per Jeff Jones' year-to-date through July 2007 OS vulnerability analysis, the number of highly severe vulnerabilities fixed by workstation OS vendors is as high as 60 and the total number as high as 170. The detailed graph can be seen in Figure 1.

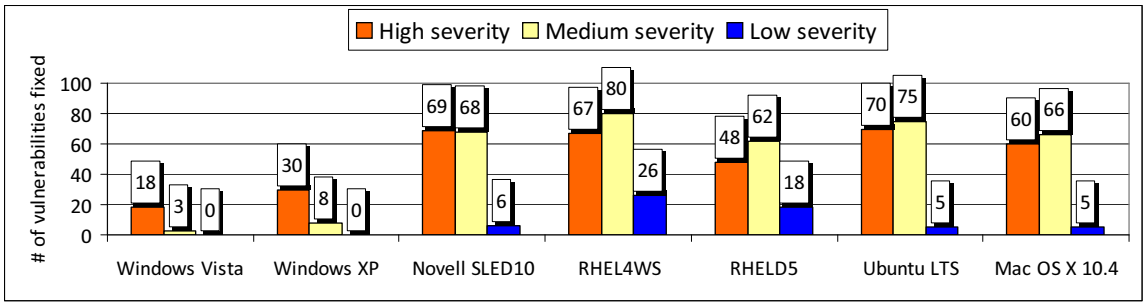


Figure 1: Vulnerabilities fixed in Workstation OS YTD through July 2007 as per Jeff Jones' report

Figure 2: Exponential growth of Rootkits as per McAfee's reports

Currently, the thrusts towards providing the kernel level security are mainly in software, either through the redesign of an OS or provide security patches to reduce the number of known vulnerabilities. Yet designing a monolithic operating system with zero vulnerability is practically unrealistic due to the growing complexity and its sheer size, often in many million lines of codes. On the other hand, a large amount of research activities are invested in designing intrusion detection systems that help in detecting malware after attacks took place.**(***HHL: Be careful for this sentence since you are also doing the post-attack thing rather than preventing it from happening.)** Although these techniques are partially successful in achieving a secure system, defending against malware in the software stack is neither a proactive nor an effective solution to achieve desirable security guarantee. This is because, the vulnerable software stack is becoming a common battle ground for both malware and intrusion detection systems. Both run in the same stack trying to counteract each other. In fact, it is always a losing battle for legitimate solutions/patches as it is easy for the malware to attack knowing the defense mechanism of the system. This results in a never ending loop to provide new software solutions for emerging threats attacking different vulnerabilities. Once the kernel is compromised,

the whole defense mechanism of the system fails and the corrupted software stack will be able to manipulate the entire underlying hardware resources at the disposal of the adversaries. The hardware cannot distinguish a legitimate from other malware processes and becomes merely a slave to the compromised kernel. Hence 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 positively identify. **(***HHL: Need to insert a sentence describing all the potential damages, such as keylogger to steal password, and such.)** After an initial kernel level attack, the Rootkit installs itself and hides all the spawned malware processes that make use of the hardware resources without leaving any trace to the system administrators. Rootkits achieve this by manipulating the compromised kernel to hijack all the utilities used by system administrators to know the state of the system. Once this attack succeeds, malware applications will be free to exploit the entire system with whatever they want without limitation. The most important objective of a Rootkit is to achieve stealth and to hide malware applications running on the compromised kernel. As per McAfee's reports, there is an exponential explosion in the number of Rootkit techniques recently. The detailed graph can be seen in Figure 2

Despite there were a lot of research attempting to address the security issues in the operating systems. Nevertheless, little attention is given to the hardware support for enhancing OS security.**(***HHL: As I said, this statement is too bold. We need to discuss why all previous schemes do not apply to rootkits. Revise the previous sentence and add more.)** We do a comprehensive study about kernel Rootkits, current solutions proposed and their weaknesses. Seeing the need to have a hardware solution, in this paper, we propose architectural support to have a direct feedback path between the hardware and the system administrator by-passing the compromised kernel. This direct-path helps the system administrator to know what processes are running on the hardware without querying a compromised list of processes that the kernel maintains. **(***HHL: Before jumping into your proposal, you need to summarize what particular problems you are addressing.)** Our proposed architecture contains the following novel components:

- ***Process Identifier Registration***: Aside from the page directory base address, current hardware does not keep additional information regarding the software context running on it. It just needs the set of context registers, program counter, and the virtual memory information to execute a process. There is no authentication before the hardware

executes a process. Process authentication is one of the novel components of the proposed architecture. A Hardware Process Identifier is introduced that has to be loaded with the software maintained PID before running any process.

- ***Address Space Encryption***: **(***HHL: Consider to merge this and decryption, anyway, require some major rewrite after we are done finalizing the whole scheme.)** We propose the use of process address space encryption to prevent the exploit of kernel rootkits. The system encrypts the physical page numbers and stores the encrypted version in the page tables. This Encryption is based on the Hardware Process Identifier (HPID) of the process whenever the new page table entry is created. This makes the OS know only the Encrypted address spaces and not the actual physical locations by reading the page tables. This kind of hardware virtualization will help hardware to have control over the executing processes.
- ***Address Space Decryption***: Address space decryption becomes necessary to decrypt the encrypted page table mappings to the correct physical addresses. This decryption is again based on the HPID of the process that wants to execute. This decryption is done before loading the mapping into the TLB. Once a process is authenticated, its mapping is placed in the TLB for it to execute. The process cannot execute till it is authenticated.
- ***Stealth Checker***: The Stealth Checker is run upon every context switch, which captures HPID and queries the compromised OS to know the software point of view. The idea behind this is to query the compromised kernel after knowing the fact about the processes running on hardware. If the OS tries to conceal any information, it implies that there is some suspicious activity and hence deduced that the OS is compromised.

The rest of this paper is organized as follows. Section 2 overviews Rootkits, its nature of stealth and the existing anti-rootkit solutions. Section 3 introduces the proposed architecture SHARK and the background work performed. Section 5 explains the results of architecture SHARK and Section 6 concludes.

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. **(***HHL: Need to give a more formal definition of rootkit. Not all rootkits were malicious. Given the audience are architects, more explanation will help. Also, discuss the persistent rootkit vs. in-memory rootkit, and how anti-rootkit software can or cannot catch them and such.)** They gradually

receive more attention these days as they are becoming serious security threats to all classes of computing including desktop users and machines used in server farms. **(***HHL: the following 2 sentences are not nicely framed. I would simply mention how backdoor was typically opened for rootkit, instead of saying we don't care how. Vulnerabilities used by these initial exploits are not addressed in this paper. Solving these vulnerabilities being a responsibility of OS security research, we study only the after effects of these kernel level attacks.)** Once a rootkit is installed by the malware, it provides ample amount of time and freedom for the adversaries to execute its malicious applications in stealth. A notorious example is the Sony rootkit incident in 2005. The Sony rootkit installs a hidden software without end-users' agreement when the user starts the provided XCP-Aurora to play an extended copy protected CD at the very first time. The original purpose was to monitor and prevent piracy, which is performed in stealth mode by concealing all processes with names prefixed sys. Unfortunately, the same rootkit also exposes serious security vulnerabilities via installing an unwanted ActiveX, again unknown to the user, which could invite other viruses or excute any arbitrary code from the Internet. In the next few sections, we classify different types of rootkits followed by the existing anti-rootkit techniques today and their insufficiency in 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, files from utilities that system administrators use to detect suspicious activity. This hidden malware activity can be disastrous as the entire system security is lost. There are two types of rootkits: the *user mode rootkits* and the *kernel mode rootkits*, operating in the user space and the kernel space respectively. The kernel mode rootkits are more detrimental as they can obtain unrestricted accesses at the root privilege level, thus can freely manipulate any component of the system via the compromised OS. The following techniques are commonly used by rootkits to achieve malware's stealth and subvert the system being attacked:

- **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 with access to this table can modify the SSDT entries, replace a function pointer with an address of its own, and hijack the system. This is a very popular kind of hooking that can be easily accomplished by any Loadable Kernel Module (LKM). Most of the earlier rootkits

used this technique to intercept system calls and filter out data. As all the utilities used by the administrators perform system calls to obtain the system state, thus 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 and steals secrets, e.g., passwords, social security numbers of 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 certain linked list maintained by the OS to represent active processes on the system. The utility tools of the system administrators only observe this modified linked list and will not show any sign of unintended use of computing resources.[1] This technique is very hard to detect because it is very difficult to track changes in the OS data.**(***HHL: Is the last sentence true? Form Joanne's "Concepts for the Stealth Windows Rootkit" while paper, she mentioned a tool called klister for windows to detect this easily. I am not sure how hard this is on a Linux.)**

2.2 Sophisticated Rootkits

In a technique called *Virtual Memory Subversion*, the rootkit fakes memory reads from the integrity checking utility. **(***HHL: should we mention Shadow Walker?)** The pages used by malware are marked as non-present in the page table and the page fault handler is hooked to intercept access and redirect to the malware pages. In the page fault handler, the malware checks to see whether the faulting address and the program counter are the same to differentiate between a memory execute and a memory read/write operation. If it is a non-execute memory access, the malware understands that some integrity checker in the system will try to read the malware pages and hence redirects it to un-modified original contents that are stored in another page to fake the memory read/write access. If it is a memory-execute operation, the malware gives the mapping of the page where the malware code is resident. **(***HHL: I think

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

this paragraph is not too clear regarding the attack model.)

In another conceptual complex rootkit called *SubVirt* [5], the rootkit is installed as a virtual machine beneath the host operating system 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 OSes that are protected from the original host which run many malware applications completely isolated from the original host OS. Another conceptual rootkit called *Blue Pill* [7] makes use of the advanced hardware-assisted secure virtual machine (SVM) support in the recently proposed AMD-V technology, and is claimed to be similarly applicable to the Intel's VT-x technology. Unlike SubVirt, BluePill is capable of installing a thin hypervisor *on-the-fly* and moves the host OS to become a guest OS. It becomes extremely challenging to detect them using any off-line detection mechanism.

2.3 Present Anti-Rootkit Techniques

All the anti-rootkit solutions proposed up-to-date are reactive. They try to detect an infected system and do not try to prevent the execution of such malware in stealth. **(***HHL: But how about yours? This is a strong criticism to existing technique. But you have to differentiate why yours is better than prior ones.)** The current software and hardware solutions are explained in the following sections.

2.3.1 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 comprise a *finger-print* that is unique to a particular rootkit. This technique is effective to detect the presence of public rootkits with known signatures. Nonetheless, it does not detect unknown rootkits and hence not robust enough.
- **Heuristic/Behavioral detection:** In this approach, deviations in the expected normal system behavior is used to identify potential rootkits. Patchfinder **(***HHL: reference?)** works based on the observation that a rootkit should inject additional, spurious code to the filter out results, which will increase the execution time and the number of instructions. Given this rationale, it makes use of instruction count to detect the presence of rootkits. The drawback is that false positives are often generated 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, Klister, Blacklight and Strider Ghostbuster are some of the popular anti-rootkits that use this technique.
- **Integrity based detection:** This approach compares the current snapshot of system memory with a trusted baseline. Difference in comparison is taken as an evidence of malicious activity. Tripwire and System Virginity Verifier 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. Note that the compromised software/OS can easily disable or circumvent these anti-rootkit software detection mechanism. This also 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 and BluePill make it much more difficult for such software anti-rootkit techniques to be effective. The ideal way of executing such anti-rootkit techniques is to have an isolated environment where the system's integrity is checked regularly and is not sitting within the same compromised stack.

2.3.2 Hardware Anti-Rootkit Techniques

The Copilot hardware detection scheme aimed at providing a more reliable, OS independent hardware solution. Knowing that an effective anti-rootkit solution has to be executed in a remote, isolated environment inaccessible to the compromised machine, they proposed this scheme. It relies on the hardware-based RAM acquisition to check the integrity of RAM by a co-processor. A snapshot of system memory is sent through the PCI bus to a co-processor where the system's integrity is continuously checked. This remote execution environment proved to be very useful as the integrity checker is completely isolated from the compromised OS and cannot be manipulated. An attack against this system was demonstrated by Joanna Rutkowska. It creates different views of the system memory to the CPU and the PCI device to subvert this hardware solution. **(***HHL: We have to discuss here or later about how ours can be immune from this type of attack.)**

3 Exploring Architectural Solutions

As none of the solutions proposed till now are strong enough to fight back against rootkits, we saw a strong need to propose a new solution. Also we know that any amount of software protection is not enough to defend the system against rootkits. This makes it necessary to come up with an OS independent hardware solution against rootkits. Rootkits being a large problem, we concentrate only on stealth achieved by kernel level non-persistent rootkits. Seeing the problem from hardware's perspective, we thought that if the hardware has the capability to identify contexts running, 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. Even though the objective is simple and clear, yet we found that it is very difficult to maintain a secure hardware list if the OS is subject to be compromised. There can be a large number of processes and maintaining a complete list of PIDs in the hardware is also impractical. The other challenge that we need to address is the mechanism to comparing the software and the secure hardware list. A potential new vulnerability could be introduced if the system is designed to ask the untrusted software to update the hardware with its (manipulated) list of processes. If the comparison is performed in software, then the exception handler must be protected or should be very simple without introducing further vulnerability. Having all these constraints in mind we studied different schemes that can be employed and discussed their weaknesses before we introduce 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. Using these PIDs in the TLB, a master hardware list of processes can be created.**(***HHL: What do you mean by master hardware list?)** This master list can then be compared with the manipulated list returned by the OS to detect the presence of hidden processes. In the beginning, this appeared to be very concrete and 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 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 to the TLB. Note that using a different PID does not affect the correctness of the malware's execution. To circumvent the mismatched 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 which is our target platform in this paper, 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 Page Table Base Address

In x86-based architectures, upon each context switch, the page table base address (PTBA) will be loaded into the CR3 register. **(***HHL: I doubt PTBA is a standard acronym.)** This CR3 register makes context switch simpler by just changing the pointer to the corresponding page table of the process being scheduled for execution. On a TLB miss, a hardware-assisted page table walk uses the CR3 register to look for the virtual-to-physical mapping in its own page table. Since each PTBA is unique for a process, one can consider using it to create the master list of active processes to detect rootkits. On each context switch, when the CR3 register is loaded with the respective PTBA, an exception can be triggered for the OS to query which process owns this particular PTBA. If the OS does not see any process with that PTBA, it implies that the process is being hidden, indicating that the OS has been compromised.

At the first glance, this approach seems to be securer than the tagged TLB scheme as the malware process cannot use the PTBA of a legitimate process, otherwise, it will not execute correctly due to the incorrect virtual-to-physical mappings. The dependency between the CR3 register and the execution context of the malware process appears to make the security scheme strong. Nevertheless, again, it is not very difficult for the compromised OS to circumvent this problem. One simple workaround the OS can perform is to swap the page tables between the malware and the victim legitimate process before the malware is ready for execution. Once this is done, the malware process can use the PTBA 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 PTBA of the malware's process.

4 SHARK: A Process Context Aware Architecture

(*HHL: Where did we mention TLB uses plaintext ... Second, it is not clear if we are only protecting Instruction side?)**

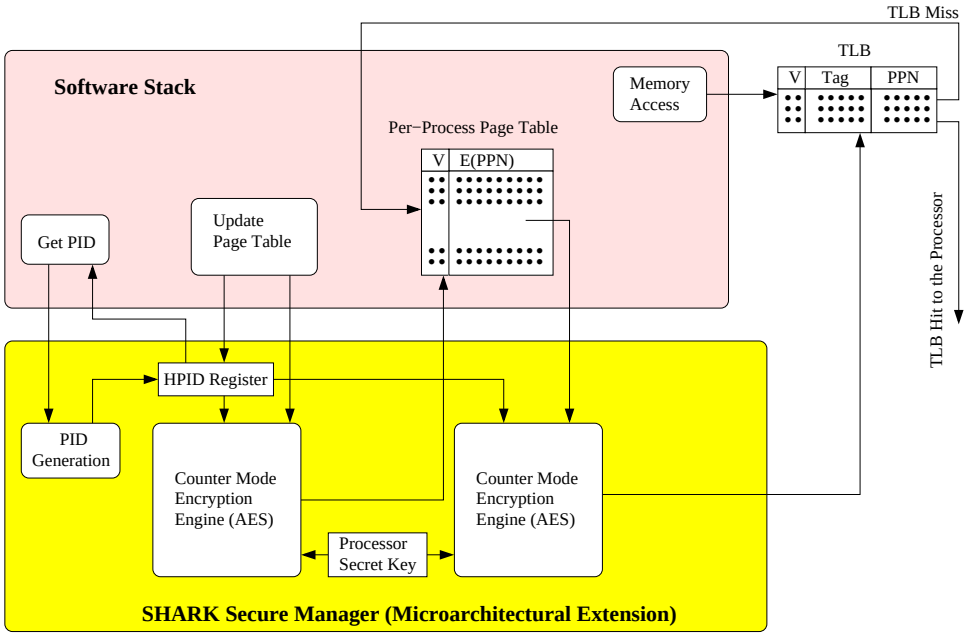


Figure 3: Architectural Support for SHARK Processor

After exploring a comprehensive, possible architectural solutions, it is evident that all of the shortcomings were due to the tightly-coupled dependency of these mechanisms with the OS which could already have been compromised. Therefore, to detect rootkits with a direct OS's intervention will always fail. To address this issue once for all, we saw the need of designing a processor architecture that is process context aware, i.e., to add minimal support for process management in the hardware. The rationale behind our ideas is (1) to 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. Making use of such hardware that can identify all the software contexts, system administrators will be able to effectively combat against rootkits that operate in stealth. The information about processes can be directly obtained by the hardware without having to rely on a vulnerable software stack. The process information obtained from hardware can be compared with the information revealed by the OS to detect whether any malware process is concealed by the OS.

Toward these objectives, we propose a novel processor architecture called *SHARK*, which stands for Secure Hardware support Against RootKit. The basic ideas behind our anti-rootkit scheme is to delegate the master control of processes to the hardware in order to enforce the security of process contexts. The OS will simply carry out the regular operations, e.g., TLB lookup, page translation, etc., under SHARK hardware's supervision and assistance. Figure 3

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 the following mechanisms into a processor system. They include

- *Hardware-Assisted PID Generation*
- *Process Page Table Encryption and Decryption*
- *Process Authentication*

The following sections detail each of these components in a SHARK processor.

4.1 Hardware-Assisted Process Identifier Generation

Conventionally, the PIDs of all the processes are generated and maintained solely by the OS. As discussed earlier, the software-generated PID values are vulnerable and subject to be exploited by rootkits to conceal malware's identity. With such schemes, it is impossible to achieve a secure mechanism to revealing all active process contexts within the compromisable software stack. Seeing this deficiency, the first component we introduce in SHARK processor 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 needs not to be a secret. Whenever the process and its page table are being created, a new 64-bit PID value for the newly created process will be assigned directly by the hardware. The size of the PID value: 32-, 64-, or 128-bit, will not affect the security strength of our scheme, which will be elaborated 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 represent the currently running process. In essence, this is similar to loading the page directory base into the CR3 register in the x86 architectures upon context switches. The HPID is an integral part of our proposed encryption/decryption mechanism to be described in Section 4.2.

4.2 Process Page Table Encryption and Decryption

The proposed hardware-generated PID and its associated HPID register in the hardware are not the solution for rootkit prevention or detection. Just by generating the PIDs in the hardware and asking the OS to load the HPID register

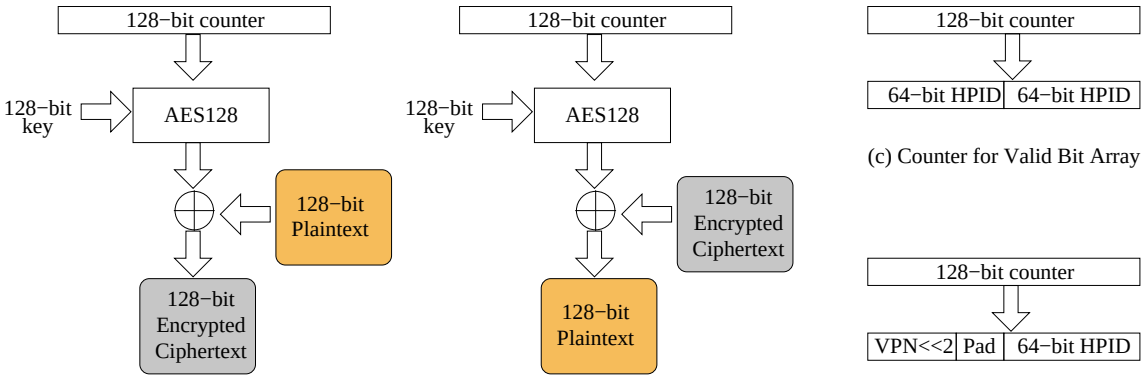


Figure 4: Counter Mode Encryption and Decryption

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 misdirect the hardware and achieve the goal of concealing malware's presence. What we need is to establish a *dependency* through 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 malware's identity will be revealed. This dependency between the HPID and the execution of the process is achieved by *Process Page Table Encryption*. Using this scheme, the entire page table of each process kept by the OS will be encrypted in a unique way proposed below using a counter mode encryption hardware in the SHARK Security Manager illustrated in Figure 3. 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 [2]. It uses a block cipher (e.g. AES [3]), 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 bitstream is generated as shown in Figure 4(a). This 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, then restores the plaintext as shown in Figure 4(b).

Counter mode is known to be secure against chosen-plaintext attacks, meaning the ciphertexts hide all partial information about the plaintexts, even if some a priori information about the plaintext is known. This has been formally

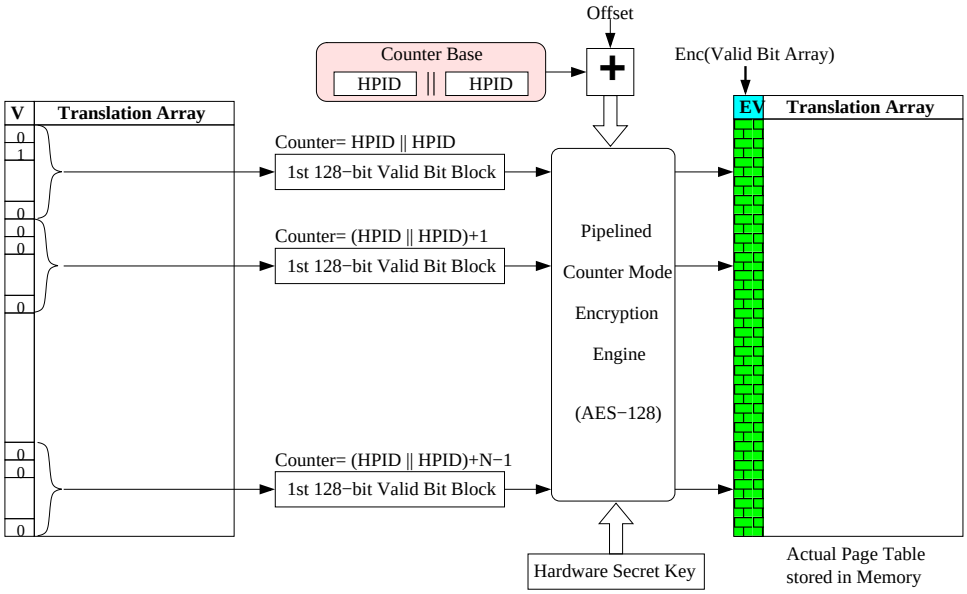


Figure 5: Valid Bit Array Encryption

proved in [1]. 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

(*HHL: Have not discussed why separate the encryption of the valid bit array and the data)**

After a new process is created and its HPID is received from the SSM, the page table will be encrypted in 2 steps for the very first page fault. Firstly, we implement a hardware secret key which cannot be read out by any means similar to what was used in prior secure processor schemes [6, 8, 9]. Using the HPID 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 page table using the encryption engine in the SSM. It is noteworthy to point out that modern processors support mixed page sizes, for instance, Intel's processors can mix pages of 4KB and 2MB (or 4MB) depending on the frequency of usage. Therefore, the encryptions need to done for all page tables of different page sizes.

We propose to concatenate the HPID value of a process, shown in Figure 4(c) as the counter value for encryption. This guarantees a different encrypted valid bit array for each individual process even though the very first page fault can

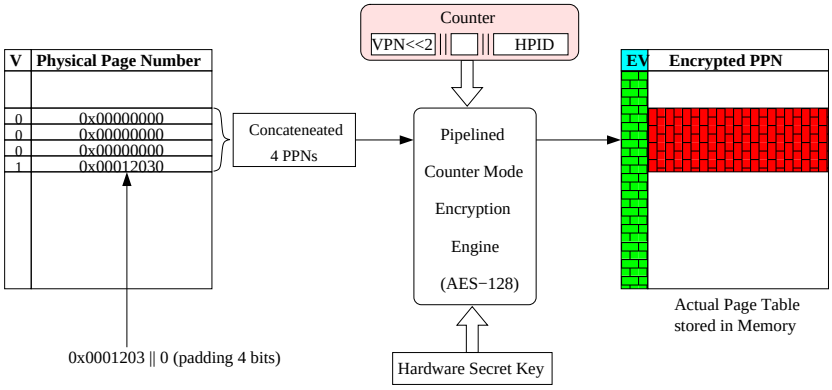
trap upon the same virtual page number (VPN). The counter value will then be incremented by one for each encrypted block starting at 0 from the first 128-bit block. The entire encryption process is illustrated in Figure 5. The result is an encrypted bitstream stored in the original valid bit structure of a page table. The significance of this initial valid bit array encryption is that from which 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 for obtaining page allocation information.

4.2.3 Page Table Translation Encryption and Updates

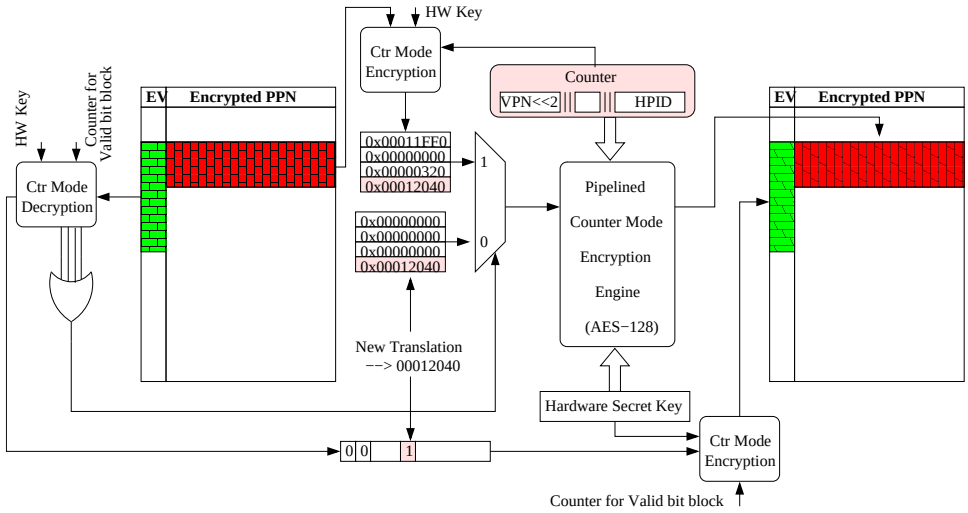
Secondly, the content, i.e., the translations of the page table are encrypted. Again, we employ the counter mode encryption and use a counter value as shown in Figure 4(d). This counter is a concatenation of the VPN and the HPID with padding when necessary. Similar to the counters used in Section 4.2.2, there is no memory overhead and extra lookup logic involved for storing and looking up the counter values since 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. Using an x86-64 processor (AMD64, Intel EM64T, or Intel 64), the maximum of 40-bit physical addresses is currently supported when PAE is enabled in IA-32e mode [4]. Given a 4KB page, the Physical Page Number (PPN) will be 28 bits. According to AES standard, the standard block size is 128 bits. Hence, we propose to encrypt four consecutive page table entries (PTEs) in the page table. It also implies we need to pad 4-bit null data for each physical address translation to make it up to 32-bit, which incurs about 13% memory space overhead for each allocated page entry. Given multi-level page tables (2-level in x86) are always used to keep the actual allocated page table small, we anticipate this overhead needed for security will unlikely be a bottleneck. Figure 6(a) shows the encryption process. **(***HHL: Did we over-simplify the scenario? We need to support different page tables, different modes such as 32-bit, 36-bit, 40-bit addresses in IA-32. How to do that needs to be discussed here.)**

When the OS needs to update the page table for new page allocation, additional work need to be performed within the page fault handling mechanism. Since everything in the page table is encrypted, to insert an translation into the page table, the SSM first decrypts the corresponding 128-bit valid bit block containing the target VPN using the correct HPID. Next, the SSM will examine the 4 entries that contain the target VPN. If none of them is valid, the SSM can



(a) Encryption Process



(b) Decryption Process

Figure 6: Page Table Encryption and Decryption

simply set the valid bit, re-encrypt the 128-bit valid bit block, and encrypt the assigned PPN with its neighboring 3 invalid mappings. However, if any of the other 3 are valid, it implies the 128-bit Encrypted PPN (EPPN) contains some valid address mapping. To maintain the correctness, the SSM has to retrieve and decrypt the EPPN, adding the new translation, and re-encrypt the EPPN to include the new translation. Figure 6(b) details the procedures. This process is required for every page table update request from the OS. In other words, all the page table updates by the OS will completely be managed by the SSM.

4.2.4 Implication to the Inverted Page Table

(*HHL: I am confused by the entire section, have done nothing here)** In the case of PowerPC 604 architecture, it uses Inverted page tables to reduce the memory overheads. It uses a single shared inverted page table that stores 40 bit VPNs and the PID of the process for which the mapping is valid. If the architecture uses an inverted page table, SHARK proposes to encrypt the stored VPNs itself. Before the mapping is loaded to the TLB, Encrypted VPNs are decrypted based on the encryption key generated. In PowerPC 604, 40 bits of VPN is padded with 24 zeros and we have a working set of 2 that uses two consecutive PTEs to generate the 128 bit input block to the AES engine. In case of PowerPC 970, that uses 52 bit VPNs, we pad 12 zeros and use two consecutive PTEs for encryption and decryption. The other obvious question is about the decryption overhead because of the associative search in the inverted page tables. The PowerPC page tables inherently uses *Hashing* to reduce the number of searches compared to fully associative search. This hashing is based on the PID of the process and the VPN of the generated virtual address. With this kind of hashing, the number of decryption required in SHARK is less. Also the decryption can be carried out only if the PIDs match first that reduces the number of decryption even more.

Please refer to Figure 7 that shows how the page table updates are handled by SSM as explained above.

Let us know consider how a memory access is handled by the SSM. Before doing a context switch to a new process, the OS loads the PID of the respective process into the HPID register. Note that this PID is not anymore just a number used by software but it is the part of the counter that was used to encrypt the page tables of the respective process.In x86 based architectures, the PTBA is loaded into the CR3 register which is used by hardware page table walks when necessary. Also on write to CR3 register, the TLB is flushed. When the process accesses a memory location, this misses the cold TLB and a hardware page table walk is performed to obtain memory mapping. As the page tables are now encrypted, the VPN to EPN mapping in the page table has to be decrypted to the right Physical Page number before accessing the memory location. In the proposed SHARK architecture, before loading this mapping into the TLB, EPN is decrypted to the respective PPN. When there is TLB miss, the page tables are walked through to get the Virtual Page Number (VPN) to Encrypted Page Number (EPN) mapping. The SSM decrypts the Page Table Entry (PTE) by first decrypting the corresponding 128 bit VALID bit array and then decrypts the EPN to PPN to get the physical location of the page. The counter values used to decrypt the VALID bit array is again the HPID register contents. The counter value used to decrypt the EPN is the HPID register concatenated with the VPN. Once EPN is decrypted, the obtained PPN

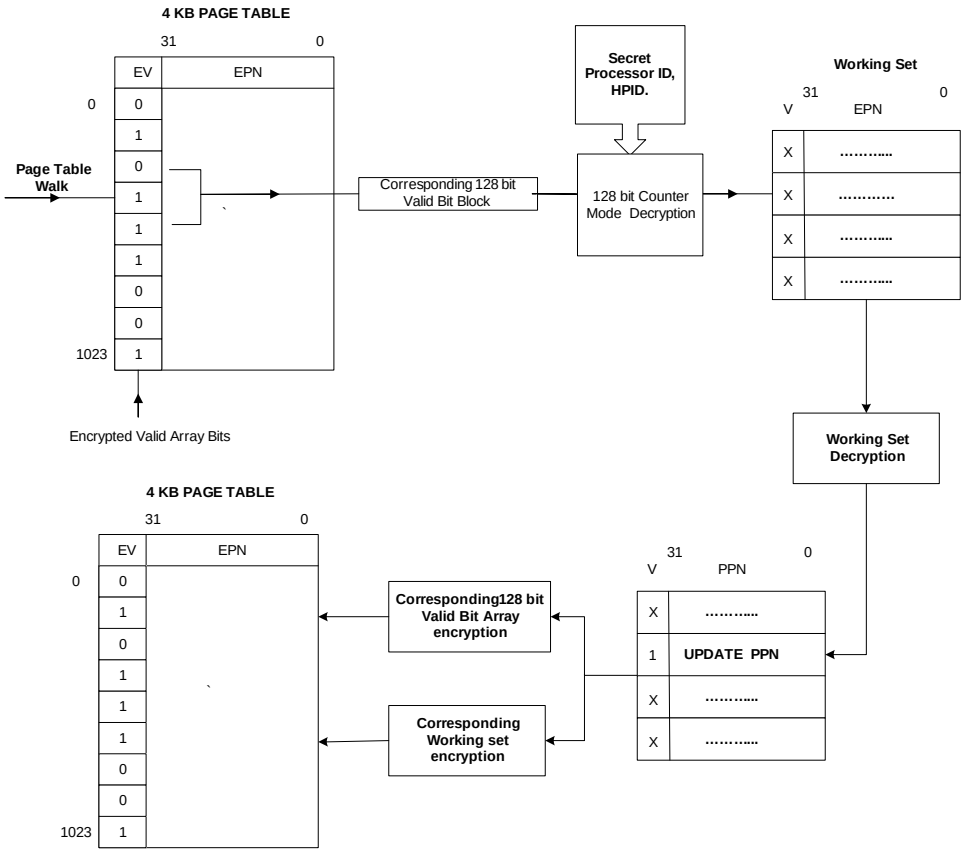


Figure 7: Page Table Update handled by the SSM

is loaded into the TLB. As page tables of every process is encrypted, the OS has to load the same HPID that was used while creating the page table mappings in order to decrypt 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 wants to execute.

To detail about the decryption technique, the EPNs are read from the all the PTE entries in the working set to generate the 128 bit input to the decryption block. This is decrypted and the required PPN is obtained by appropriately masking the output before loading the TLB with the proper mapping. Th TLB update happens as shown in Figure 8

The TLB now contains the actual VPN to PPN mapping. Once the TLB is loaded with a proper mapping, the process executes like in any other normal system.

4.3 Process Authentication

In 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. Using which, the malware cannot execute

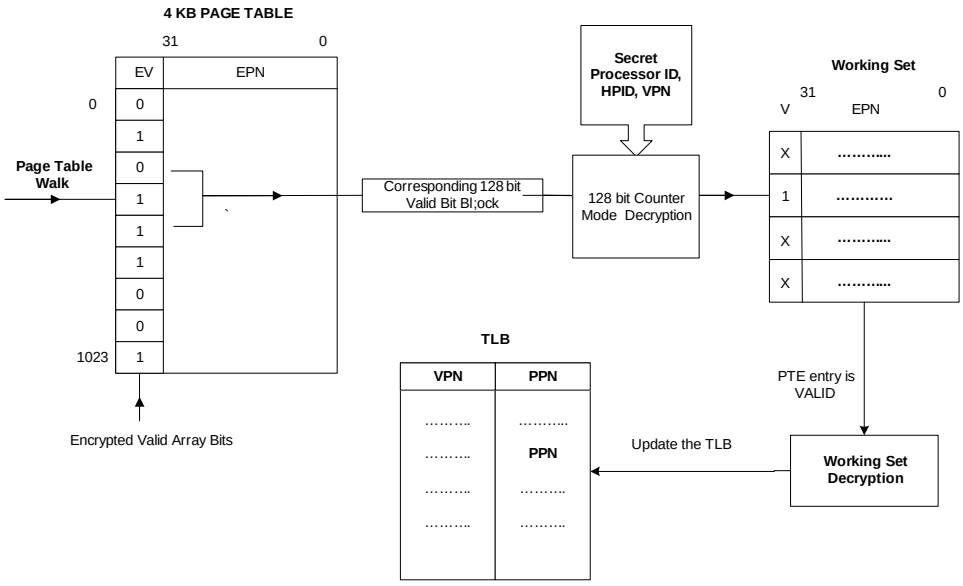


Figure 8: TLB update handled by the SSM

successfully. The proposed page table encryption and decryption are novel ideas using which we have one more level of virtualization provided by the secure hardware for the OS, putting all processes under examination by SHARK. This changes the relationship between the Hardware and OS, giving hardware the capability of controlling and authenticating the execution of software contexts through address translation process. The whole scheme of PID generated 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*. Using this Authentication mechanism in SHARK, the hardware can identify different software contexts making use of the hardware resources, most importantly, the malware running in stealth.

4.4 Stealth checker

The last component of SHARK is the *Stealth Checker*. Designing a trusted passage between the hardware and anti-rootkit software is very crucial to catch processes running in stealth. With the SHARK hardware support, now we have the information of every running processes controlled and revealed by the hardware. The final action to rootkit detection is to compare the running process context with the software list returned by tampered utilities, and evaluate the differences.

For this operation to be secure, tamper-free, and effective, again, the system has to prevent the compromised OS from intervening maliciously. One technique is to perform such comparison during each context switch. Upon every

context switch, an exception will be handled by an exception handler that retrieves the content of the HPID register to learn the upcoming process to be executed. Then, the compromised OS is queried to report the details of this process. The idea behind this is to call the same utilities that are compromised by the rootkits for obtaining the manipulated process list. When the process scheduled to be running cannot be found in the manipulated list, a rootkit detection will be signaled. Given the rootkit cannot differentiate between the system administrator calling the utilities and the exception handler calling these utilities, the hidden process will be revealed.

There are a few security implications regarding the exception handler. The obvious question arisen is regarding the safety of the exception handler itself. As the exception handler is managed by the OS, it always has access to this handler and can hijack it easily. To address this issue, we propose to have this exception handler in the *firmware*. The only possible action that the OS could do is to upgrade the firmware that requires a system restart. As the rootkit's main aim is to hide its malware's activities, it cannot upgrade this firmware on the fly to restart the machine which will indicate suspicious activity. Also, as we are dealing with memory-based, non-persistent rootkits, the rootkit will not be able to survive and sustain a system restart.

Another main concern is the communication between the firmware and OS utilities. For this purpose, we propose to use *socket programming* to enable communication between the OS socket and the firmware. The firmware requests the OS to open a new console as if it were to be a remote user trying to open a console. There is no way that the OS can distinguish these requests and have to service them . Once a console is open, the firmware can just send commands to the console to execute on the compromised host. These commands directly use the compromised utilities such as using ps -p <pid> to query for a particular process with HPID as its process identifier. Also, the output of these commands can be sent to a file on disk which the system administrator can check later to scrutinize any hidden activity. Yet another option is to send the output to a remote machine that is isolated from the compromised OS.

(*HHL: I do not understand this paragraph.)** In another simpler approach, on every context switch, the HPID can be captured by the exception handler in firmware and sent to a remote machine through the PCI bus to log the running processes continuously. The system administrator can analyze the unmodified process list on the remote machine. Also the system administrator can connect to the compromised machine remotely and execute the system utilities like ”ps” and obtain the list of processes that the compromised OS reveals. This list can be compared with the process list that it obtained from the firmware which a golden copy. Any mismatch reveals that malware stealthy

processes are running on the machine. The technique can be easily automated to detect hidden processes.

4.5 Strength of SHARK

This section discusses potential threat models and analyzes the strength of a SHARK processor. Knowing that the operating system 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 hijack a legitimate process' PID instead of its own to conceal the PID of the malware process. As analyzed previously, this will cause the malware process fail to execute as it cannot decrypt the address mappings correctly. Note that, the encryption is seamless established using the HPID at the very first time a page fault is encountered.

As the crux of SHARK architecture is using the PID of the process to encrypt and decrypt the page table. The rootkit may plan to encrypt the page tables of a malware process using the hijacked PID of a legitimate process. As such, the process can always use this legitimate PID to obtain correct physical addresses. This attack will also fail because the page tables are encrypted using the HPID generated by the 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.

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 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 which 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.

Last but not least, one may wonder why the compromised OS cannot simply updated 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 secret key, which is burned in and cannot 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.

Rootkits are getting more sophisticated these days and they are exploiting every vulnerability to get into the software stack and start executing malware processes in stealth. After compromising the system, they mainly want to execute their malware applications on the system as long as possible without disturbing the running legitimate applications and also not leave any trace to the system administrators. The recently developed rootkits such as BluePill [7] and SubVirt [5] move the host OS to become a guest OS on a virtual machine monitor installed by the rootkit. Once this is achieved, nothing in the software stack can be trusted. Our SHARK architecture will become more useful when the complexity of these rootkits increase as it does not solely depend on the software stack. It is not possible to address every vulnerability in software. SHARK does not care about the nature of software attack which could occur at a user level, kernel level or hypervisor level to conceal the malware applications and abuse the computing resources as well as compromise data confidentiality. As SHARK is assisted by hardware and does not depend on the software stack of a compromised machine, it will be able to detect these stealth malware applications irrespective of the software vulnerability exploited by the rootkits.

5 Experimental Results

We conducted two sets of experiments to evaluate the proposed SHARK Security manager (SSM). In one set of experiments we evaluated the strength of the proposed scheme against malware running in stealth. In the other, we conducted experiments to evaluate the performance overhead incurred by using the proposed SHARK architecture.

As a proof of concept, we installed rootkits on Linux OS running on a simulated SHARK architecture. SIMICS was used to simulate the SSM module. Linux Kernel versions 2.2.14 and 2.6.16.33 were modified and recompiled to use the SSM simulated in SIMICS. The following rootkits were inserted into the base kernel as Loadable Kernel modules(LKMs)

- Adore 0.42
- Knark 2.4.3
- Phide

- Mood-nt-2.3
- Enyelkm.en.v1.1

These rootkits which are inserted as kernel modules have access to the kernel space and they modify the system call table, interrupt descriptor table to change the execution flow of the compromised operating system and provide utilities to hide malware processes from system administrator utilities like ”ps”, ”top”. Using such a set up, we had a compromised software stack and used SHARK architecture to catch these hidden processes.

The scheduler of the base kernel was modified to load the HPID register with the PID of the process before context switching to the next process. Every write to this HPID register was handled by an exception handler that was serviced by the SSM. The HPID register was read and the same compromised utilities like ”ps” and ”top” were queried to find a process with a PID equal to the HPID contents. As these utilities were compromised, they try to conceal the information of malware processes. This implied that the processes running on hardware were not shown to the system administrator software utilities and the SSM triggered an alarm to the system administrator. This way we evaluated the functionality of Shark architecture as an autonomic hardware anti-rootkit solution that can be used to catch processes running in stealth.

5.1 Performance Evaluation

We use SIMICS to evaluate the performance of the proposed SHARK Security Manager. We model a 2G-Hz in-order processor with split L1 Data and Instruction caches and an unified L2 cache. We take a pessimistic approach here by not simulating an out-of order machine deliberately to expose the absolute overhead of the proposed scheme. In fact, in a practical scenario, the actual overhead will be hidden by using an out-of-order machine. Both the cache levels have 64 Byte Blocks and use LRU replacement scheme. We have L1 caches of size 32 KB each, 8-way set associative. To find out the sensitivity to L2 cache size, we use 2MB and 4MB sizes, 8-way set associative.The simulated encryption/decryption engine is a 16 stage pipelined 128-bit AES engine.The input block size to encryption engine is 128 bits.

We simulate two different timing configurations.In a typical timing configuration, L1 caches have a hit latency of 2 cycles, L2 has a 14 cycle hit latency. The memory access latency is 200 processor cycles. To study the sensitivity to the AES engine latency, we use 80 cycle and 160 cycle encryption/decryption latency. In an aggressive configuration, L1

caches have a 3-cycle hit latency, L2 cache has a 30 cycle hit latency, memory access latency is 300 cycles. and AES engine latency is 160 cycle and 240 cycles. Also we model 6 configurations of TLB by varying the size and associativity of 4K page and 2MB page mappings.

We use 28 SPEC2K benchmarks, to study the performance overheads of the proposed SHARK mechanism. For each simulation, the reference input set is used and we simulate the first 2B instructions. We do not fast forward before simulating. As Page faults are critical for our evaluation, we do not want to miss the early page faults. Linux kernel 2.6.16.33 was recompiled to send requests to the SHARK security manager whenever there has to be a Page Table Update. When the SSM gets control, it encrypts the PTE and updates the page table. This requires 1 Valid Bit Array Decryption + 1 PTE Decryption + 1 PTE Encryption + 1 Valid Bit Array Encryption. The overall overhead for a page table update is 4 times the latency of the AES engine. Also before adding a new mapping into the TLB, we have to decrypt the contents of the PTE. This requires 1 Valid Bit Array Decryption + 1 PTE Decryption leading to an overhead of 2 times the latency of the AES engine. The Page Table updates and the TLB updates are the sources of overhead. Also we are flushing the TLB on every context switch like in x86 machines.

Figure 9 shows the execution time overhead in percentage for all the benchmarks run for different TLB configurations. The L2 cache size here is 2 MB and we have a typical timing configuration and the AES encryption/decryption overhead is 80 cycles. We observe that TLB configurations are very critical to evaluate our overhead. Some benchmarks like lbm, bwaves, bzip2 need more 2 MB code mappings in the TLB. As we are flushing the TLB for every context switch and the number of context switches for these benchmarks lbm, bwaves, bzip2 are very high compared other benchmarks as shown in Figure 11, we are losing data in the TLB often and hence the overhead is more. By increasing the 2MB code entries in the TLB, we see that the overheads of these benchmarks fall down because of lesser TLB mappings brought into the TLB and hence lesser overhead even during a context switch. For all the other benchmarks, we see that the overhead comes down as we increase the TLB entries and its associativity.

We conducted a sensitivity test by changing L2 cache size and different timing configurations to see the effect. Average percentage overhead of all the benchmarks run on all the six TLB configurations and combinations of the L2 Cache and Timing configurations is calculated and plotted in Figure 10. The timing configurations are as shown below in Table 1. We observe that the L2 cache size does not have any impact on the overhead of the scheme. But again the TLB configurations are playing an important role to determine the overhead. Increasing the AES encryption/decryption

Table 1: Processor System Configurations

Timing Configuration	frequency	L1 latency	L2 latency	Memory latency	AES latency
TC1	2GHz	2	20	200	80
TC2					160
TC3	4GHz	3	30	300	160
TC4					240

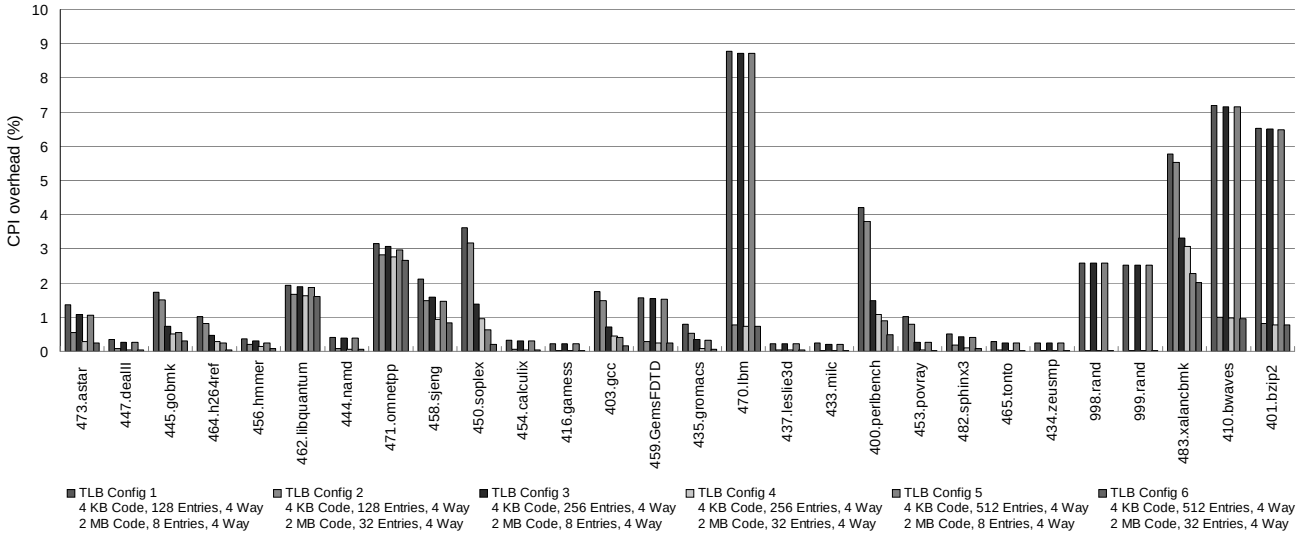


Figure 9: Percentage Performance overhead of different TLB configurations for SPEC Benchmarks

latency in the second configuration is naturally increasing the performance overhead. The best performance is observed for a very good TLB configuration and is below 1 percent for all the machine configurations and benchmarks.

6 Conclusions

References

[1] 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.

[2] W. Diffie and M. Hellman. Privacy and Authentication: An Introduction to Cryptography. In *Proceedings of the IEEE, 67*, 1979.

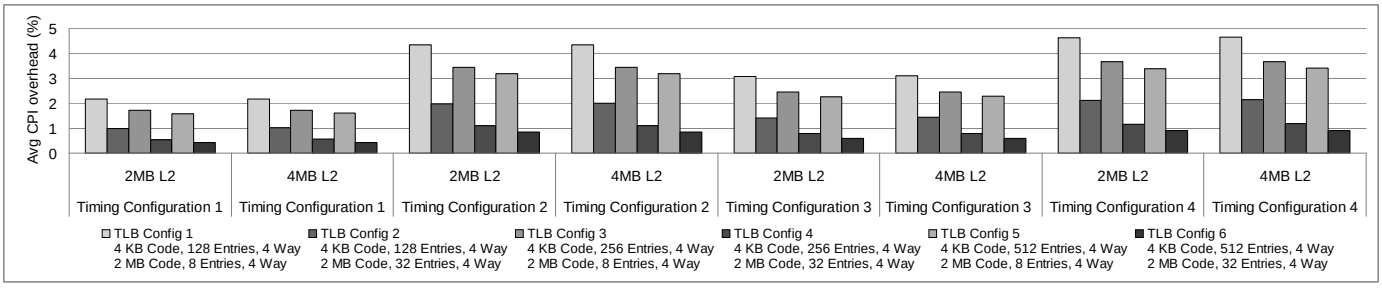


Figure 10: Average Percentage Overhead of all the benchmarks for different configurations

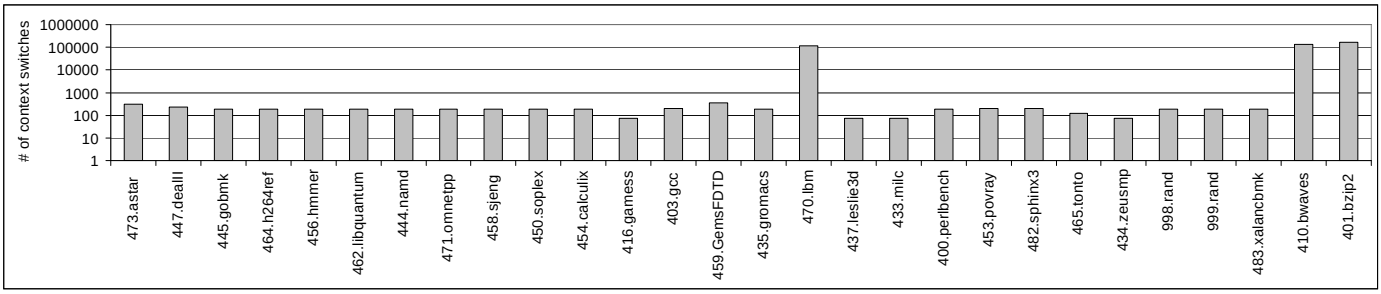


Figure 11: Number of Context switches (amid 2 billion instructions)

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

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

[5] 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.

[6] 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.

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

[8] 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.

[9] 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 Int'l Conference on Supercomputing*, 2003.