A Hypervisor-Based System for Protecting Software Runtime Memory and Persistent Storage

Prashant Dewan[↑], David Durham[↑], Hormuzd Khosravi[↑], Men Long[↑], Gayathri Nagabhushan[⊥]
↑: Communications Technology Lab, Intel Corporation; ⊥: Software Solutions Group, Intel Corporation
2111 NE 25th Ave, Hillsboro OR, 97124, USA
Corresponding author: men.long@intel.com

Keywords: computer security, hypervisor, virtual machine monitor, runtime memory, performance evaluation

Abstract
An important goal of software security is to ensure sensitive/secret data owned by a program shall be exclusively accessible by the program. An obstacle to such security goal is that modern commodity operating systems (OS) for the sake of speed and flexibility have a unified linear address space--any OS kernel program can access all the linear addresses. As a result, rootkits or malicious system software are able to control the OS virtual address space, harvest the sensitive data used by software programs on the compromised computer, and report the data to remote entities controlled by hackers.

In this paper, we present a holistic approach against sophisticated malware. Instead of focusing on the security of various abstraction layers of OS, we utilize the hardware techniques to directly provide the trust services to software programs. Without modifying OS, we leverage the virtual machine monitor technologies to create a lightweight hypervisor for fine-grain software runtime memory protection. As a result, a program's memory could be hidden from other high privilege system software in a single commodity OS. In addition, we propose the data locker component in the hypervisor, which prevents the sensitive data of software program in persistent storage from leaking to rootkits or other malware. For the performance evaluation, the implementation based on hardware-assisted x86 virtualization technology is presented and experimental results are reported.

1. INTRODUCTION

In this paper, we address an important problem in secure computing: how to continuously protect sensitive information that executing software needs, in face of possible OS kernel penetration by malware. It has been well reported that many hackers, for the financial gains, target the computers of enterprise users. For example, a hacker installs a rootkit on a computer. The rootkit may read and disassemble the authentication secret of a client VPN driver that is stored on the hard drive. Alternatively, the malware could directly read the authentication secret of the VPN program from the runtime memory. Many modern commodity operating systems have the shared linear address for the kernel programs, which has the historical roots for improving the execution speed of software programs. The resultant security tradeoff is that any kernel program can access the linear address, which opens the doors for hackers to harvest the sensitive data of software programs (e.g. reading the stored file and disassembling it, directly reading/writing the memory of the runtime image of a software program, etc).

From the perspective of an efficient and effective systems security, it is advantageous that we take a holistic approach to protect both the runtime memory and the persistent storage of software program. Notice that many existing hardware-based methods take a narrower approach. For instance, security hardware (TPM or trusted platform module, etc) encrypts data using the key stored in tamper-proof hardware, which can defeat many attacks on obtaining the encryption key. However, under the attacks by malicious code with root privilege, gaps between TPM and software programs may compromise the protection of a program's secret data. For instance, hackers compromise an OS kernel by exploiting a kernel or device driver bug at runtime. Then, the malicious code can manipulate platform configuration states and trick the TPM into decrypting sealed data. Additionally, the malware with root privileges can just read or modify secret data when the data is loaded in memory for program execution.

Another important design consideration is that we focus on hardware based technique that accommodates both legacy and future software programming and operating systems. In our context, we assume that the protected program, other legitimate applications, and malware are all running in a single commodity OS. This reflects the current PC computing paradigm in which multiple programs work together with complex abstraction and modularization to satisfy the user experience. To accommodate the paradigm, we are utilizing the capability of a hypervisor's oversight of physical memory, and obtaining a lightweight hypervisor component which has the ability of "hiding" memory from the rest of the OS components. The complexity of this hypervisor is relatively small, as this security problem does not mandate a full machine virtualization. In addition, the hardware based security technique can take the advantage of the emerging hardware-assisted x86 virtualization technologies—commodity OS without changes can run "fast" on top of a virtual machine monitor based on x86 CPU and chipset with virtualization support.

A high-level view of the proposed architecture is depicted in Fig. 1. We assume the trusted platform techniques to establish the secure launch of the hypervisor. For the proposed method, the hypervisor is comprised of three modules. After OS loads a program, the integrity measurement module (IMM) measures the runtime image of the software program. If the measurement result is good, the memory address information for the protected memory region will be registered with the memory protection

module (MPM) in the hypervisor. Subsequently, the MPM enforces the boundaries of the protected software program, ensures proper control flow into exported entry points of the protected program, and prevents malware or even other kernel code from reading/writing from/to the region of protected memory. This attains the goal of "hiding" memory pages for a software program from the rest of the OS. During the initial registration phase, the sensitive data of the software program is sealed by the secure vault module (SVM) through encryption and message authentication code. Subsequently, when the protected software needs to access those sealed data during executions, the sealed blob is passed by the software program to the hypervisor. The SVM verifies and decrypts the blob, and then puts the decrypted data in the hidden memory pages of the program. In summary, the proposed method provides a holistic protection on software sensitive data (in static storage, transit from storage to memory, and in memory), even though the malicious codes can reside in the OS kernel and device drivers.

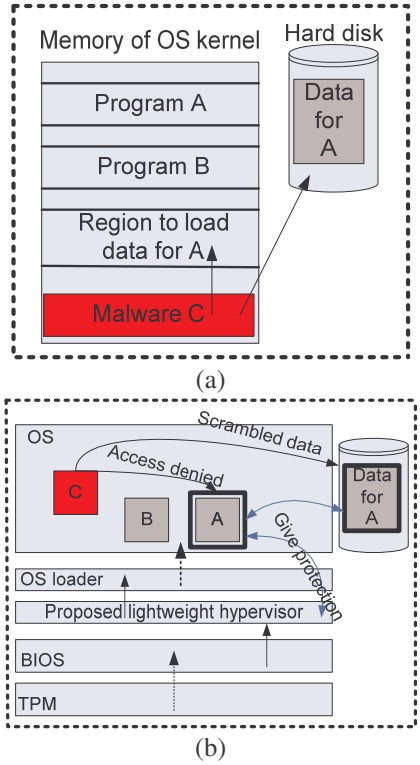

Fig. 1: (a) threat model on data leaking to malware where legitimate programs and malware reside in the same OS, (b) systems diagram on the proposed architecture. TPM (trusted platform module) measures and ensures a good launch of the hypervisor. The hypervisor can give a continuous memory protection on the runtime image by rejecting malware memory access and the protection on persistent storage through the encryption by the cryptographic key only known to the hypervisor.

The remainder of the paper is organized as follows. In Section 2, we review the background and the existing relevant solutions. In Section 3, we detail the three components of the hypervisor on memory and storage protection technology, explain the design novelty and rationales, and present the security analysis. In Section 4, we report some performance evaluation results based on our prototype implementation. Finally, conclusions appear in Section 5.

2. BACKGROUND AND RELATED WORK

2.1 Program Isolation by Virtual Machine

An important line of research has been the use of multiple virtual machines for program isolation where protected programs run in dedicated virtual machines. For instance, the Terra architecture [1] is able to provide a separate partition of a high-assurance virtual machine (VM) to host a security-critical application. In a finer granularity, the Tahoma architecture [2] (implemented on the Xen hypervisor [3]) isolates web browser instances by sandboxing them in a VM. In the sHype architecture [4], the access control policies are added to the Xen hypervisor to enable strong isolation, mediated sharing and communication between virtual machines. The main difference between the multiple-VM methods and our approach is that ours is focused on providing the intra-linear-address-space protection to the programs running in a single VM, which better conforms to existing software programming models. In addition, the proposed method could avoid the expensive VM switching for modular programs—one program could occupy one VM in extreme case.

2.2 Data Sealing

There is a sizable catalog of methods on protecting data storage. Obfuscation of executable code and its sensitive data certainly improves resistance to disassembly when hackers steal the obfuscated version of code/data [5]. However, with enough patience, assembly language expertise, and good debugging tools [6], hackers are able to learn the software sensitive data. One noticeable line of works on protecting storage is the usage model of data sealing/unsealing in TPM [7], [8] or the next-generation secure computing base (NGSCB) [9]. The TPM sealing process takes the external data, a requested PCR (Platform Configuration Register) value, encrypts them, and returns the sealed data package to the caller program. During the unsealing processing, the TPM compares the current TPM PCR value to the requested value in the sealed data package. If the two values do not match, the unseal operation aborts. Sealed storage is a similar feature in NGSCB, which provides the confidentiality and integrity for persistently stored data. Recently, hard drive encryption leveraging TPM [10] has also been gaining acceptance from consumers, as laptop theft poses a great danger to data safety. On the evolution of the hacker offensive side, a hacker, however, may successfully compromise an OS kernel or device drivers during runtime after TPM finishes the chain of trust for loading the OS. Next, the hacker code can send the blob of interest to the TPM and get the decrypted data because the PCR values still match. Another feature of TPM sealing is that any software can seal a blob for any other software [7]. In contrast, the proposed hypervisor locking/unlocking intends to solve the

orthogonal problem: ensuring that only the software programs associate with the sensitive data can ask the secure vault module to encrypt/decrypt data. Though the high-level ideas of the proposed method are similar to those in TPM or NGSBC, we design the method based on a lightweight hypervisor and, more importantly, fill the security gap between hardware and software programs. The proposed method holistically provides the runtime memory and persistent storage protection for a given software program against malware with root privileges.

2.3 Runtime Memory Protection

One important issue on protecting data storage is the realization that the application needs to get the unsealed data loaded to its memory, and the data needs to stay there for the execution. Malware with root privilege is capable of reading or writing the unsealed data, as modern operating systems essentially make available the kernel virtual address space for all OS components and device drivers. Various defense methods on runtime memory protection have been proposed in literature.

One line of research is to periodically measure the program image so that the malicious modification will be detected. For instance, the Copilot architecture [11] is to let a secure coprocessor measure the crypto hash of software runtime memory and compare to the known hash result stored in the secure coprocessor. In the SWATT [12] and the Pioneer systems [13], a challenge-response protocol coupled with memory measurement is used by a remote entity to verify the memory content in local platform. For those methods, between two consecutive measurements, there is a time window for malware to compromise some OS program or device driver. And when the modification is discovered – the damage has already been done.

On another front of defense measures, address randomization method can deter hackers from compromising program runtime memory [14], [15]. Another distinct method based on trusted computing architecture was exemplified by Cerium architecture [16], where a physically tamper-resistant CPU and a microkernel protect programs from each other and from hardware attacks. Nevertheless, those methods are limited in protecting the current common OS invariants on x86 platforms. A relevant work on isolating device drivers from OS kernel programs was reported in the Nooks architecture [17]. The usage model of the Nooks isolation manager is to recover or contain the device driver errors, though not necessarily to protect from malicious intent.

3. ARCHITECUTRE OF MEMORY AND STORAGE PROTECTION

In this section, we describe the architecture and novelty of the three components of the proposed lightweight hypervisor: Integrity Measurement Module (IMM), Memory Protection Module (MPM), and Secure Vault Module (SVM). The IMM ensures that the MPM can identify and then place protections over the runtime memory image of a validated program. The SVM provides the service of locking/unlocking data for the legitimate program that needs the data.

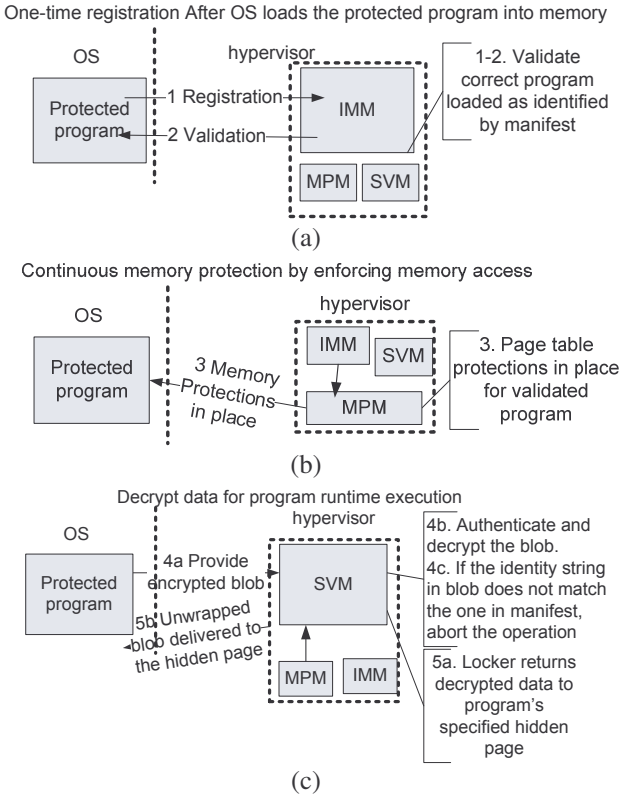

Fig. 2: Functional diagrams illustrate the hypervisor unlocking service for a protected program. The life cycle is: (a) hypervisor verifies the authenticity of the software program, (b) based on the registered memory addresses, hypervisor places a runtime memory firewall over the program, (c) hypervisor decrypts the data blob for the authentic program, and then puts the decrypted texts to the protected memory region

3.1 Integrity Measurement Module (IMM)

This purpose of this module is twofold: locating the program in host runtime memory, and verifying that the program has been loaded into virtual memory without any tampering. Extensive works have been done in device driver signing for some modern operating systems [18]. For instance, a secure integrity measurement system for Linux was reported in [19].

In contrast, this IMM design in the proposed method can measure runtime program image that might be different from the stored binary due to OS loader manipulation. The OS loader performs relocation operations on the sections of the program which modifies contents of these sections. To be able to verify the integrity of these sections at runtime by an entity other than the OS loader, a program integrity manifest must include information about reversing these modifications, e.g. external symbols and relocation entries. This manifest information is signed by a program's vendor. In addition, the manifest also includes the Program_Identity_String which identifies the program, its version number and any other relevant information to uniquely identify a particular program. For the details of the format of the program integrity manifest, we refer reader to [20].

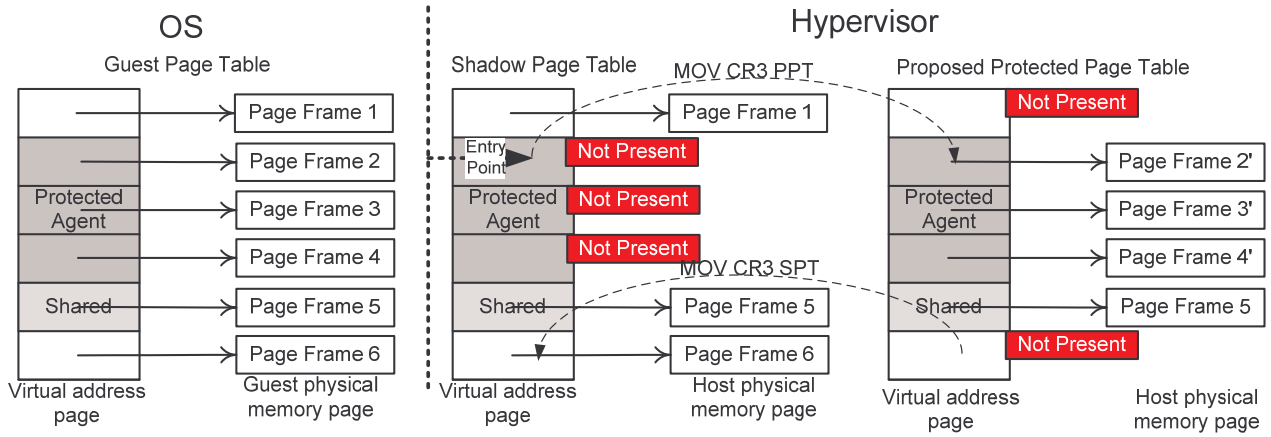

Fig. 3: The high-level view of the proposed protected page table for runtime memory protection. Guest page table is maintained by OS. Shadow page table is an inherent feature of hardware assisted x86 memory virtualization on managing multiple virtual machines. Protected page table is the new feature in this paper. PPT: Protected Page Table. SPT: Shadow Page Table. When the protected agent is accessed from the valid entry points, hypervisor will switch it to the protected page table and then execute there. If reading/writing memory does not follow the entry point, page fault on SPT will occur and trap the control to the hypervisor. Then the memory protection module can allow or deny memory access by comparing the faulting address with the registered valid memory range.

3.2 Memory Protection Module (MPM)

This security goal of this module is to prevent run-time, memory-based attacks against a protected program,

- The program's invariant portions (e.g. code and static data) cannot be tampered with,
- The program's internal dynamic data (the data that changes with execution of the code) can only be accessed by the program,
- The program's code can only be entered at pre-defined entry points to ensure proper control flow.

The proposed module relies heavily upon the underlying virtualization technologies. There are three major flavors of virtualization for x86 commodity machines: binary translation, hardware-assisted x86 solutions, and paravirtualization. We refer readers to [21] for the comparison on those different virtualization methods. In our proposed system, we utilize the hardware assisted x86 virtualization method—in particular the VT-x solution [22].

Managing host physical memory is an important task for virtual machine monitors (VMM). When a software program is loaded by an OS into the memory, the OS assigns a range of virtual addresses to the program, and makes modifications to the guest page table controlled by OS. In virtualization systems, the VMM maintains a separate set of page tables (inaccessible to the guest OS) called the shadow page tables, which are exclusively used by a CPU for translating guest virtual address to host physical address.

To protect the runtime memory against malware with root privilege, we create another set of page tables (protected page tables) to partition the virtual address space represented by the shadow page tables. The proposed protected page tables are created in a physical memory area that is assigned exclusively to the hypervisor, and there are no mappings to these physical pages from shadow page tables.

After creating the protected page tables, the memory protection module copies the entries that correspond to the virtual address range of the verified software program from the shadow page tables to the protected page tables. Next, the protection module marks all the entries (nonprotected) in the protected page tables as "Not Present" while the entries corresponding to the virtual address range of the protected program as "Not Present" in the shadow page tables. Finally, the protection module scans the shadow page tables to ensure that no other entries in the shadow page tables point to the host physical pages that contain the protected software program.

The protected page table complements the existing technique of shadowing page tables by breaking the same linear address space as seen by OS into two smaller domains. As shown in Fig. 3, page frames 2, 3 and 4 belong to the protected virtual address range of the program, and are consequently marked as "Not Present" in the shadow page tables. Any attempt by various components of the operating system to access these pages would cause a page fault to the hypervisor. Similarly, page frames 1 and 6 belong to OS components other than the protected agent, and hence are marked as "Not Present" in the protected page tables. Any attempt by the protected software program to access these page frames would likewise cause a page fault to the hypervisor. Finally, page frame 5 is shared between the OS and the protected agent, and hence has a mapping in both tables.

The memory protection module enforces separation by monitoring the page faults encountered during runtime. For hardware-assisted x86 virtualization technology [22], a page fault automatically causes a VM exit event to the hypervisor. The memory protection module in the hypervisor intercepts this page-fault. Based on the processor register values (e.g. EIP and CR2 in x86), the protection module determines the type of the memory

access (instruction fetch or data access). In case the page fault is caused by an instruction fetch (e.g. JMP or CALL), the protection module checks if the page fault is caused by a legitimate context switch event to a valid entry point of the protected agent. If the context switch is valid, then the protection module appropriately switches the page tables in the hypervisor by changing the value of the CR3 register. When malicious rootkits write/read to/from the protected region, the faulting virtual address is not in the protected range of virtual address. The protection module will thus detect and deny this memory access.

3.3 Secure Vault

When an application or system software is loaded, it will register to the integrity measurement module and also request the locking service for its sensitive data. The integrity measurement and memory protection are the prerequisite steps. Thereafter, the secure vault module can provide the service of locking/unlocking data for the protected programs.

When the running software requests for the locking service, it passes a data blob that needs to be locked by the secure vault module in the hypervisor. The format of data wrapper in terms of crypto essential is

$$\text{Data\_Wrapper} \leftarrow \text{Data} \,||\, \text{Program\_Identity\_String} \quad (1)$$

where Program_Identity_String can identify the program that requires the data for runtime execution and tie it with the program's integrity manifest. The secure vault module will check whether the Program_Identity_String that is passed from Data_Wrapper matches the one that passed the verification by integrity measurement module. Next, the secure vault module performs the crypto operation to lock the data

$$\text{Locked\_Blob} \leftarrow \text{AES-GCM}(\text{Vault\_key}, \text{Data\_Wrapper}), \quad (2)$$

where AES-GCM is an authenticated encryption mode standardized by US NIST. In practice, vault_key (an AES crypto key only known to the hypervisor) can be bootstrapped from TPM when the hypervisor is launched. As for the crypto primitive, one can also use AES-CBC for encryption and HMAC-SHA256 for the message authentication code. To complete the locking service, the secure vault module returns the locked blob to the requestor program's memory.

Fig. 2(c) illustrates the unlocking process when program is in memory for execution. After the requestor program passes Steps 1-3 of the program integrity check and memory protection, it requests the unlocking service and sends Locked_Blob to the secure vault module in Step 4a. The hypervisor secure vault module first verifies whether the message authentication code in Locked_Blob is correct. If so, the vault module decrypts the ciphertext and verifies whether the Program_Identify_String matches the one in the Steps 1-2. Then the decrypted data will be put into the hidden memory pages of the program for its execution. In other words, the requestor program can now use the unencrypted data from its hidden pages, which are only accessible by the program itself through the memory access enforcement by the memory protection module in the hypervisor.

3.4 Security Analysis

3.4.1 Security of Hypervisor

The security of the proposed locker technology depends on the security of hypervisor. To enforce the launch of an authentic hypervisor, we rely on the TPM of a PC platform to foil the insertion of a malicious hypervisor such as the one in Subvert [23]. In addition, without addressing the usage model of I/O virtualization, the proposed hypervisor could be realized by relatively fewer lines of code, which lowers the risks of the hypervisor being compromised by malware during runtime.

3.4.2 Security of Locked Blob

Attackers can steal the locked blob. As the Vault_key (a strong AES crypto key known only to hypervisor and TPM) is not known to attackers, they are not able to learn the plaintext corresponding to the locked blob. The assumption is that the locked blob itself is useless for hacker's purpose. If the attackers modify the IV or the ciphertext of the blob, the message authentication code will report an error and fail during the unlocking operation of the vault module.

In contrast to the TPM sealing (any software stack can seal data for another software stack) [7], the integrity measurement and memory protection modules ensure that only data can be locked/unlocked for the valid caller program that requires the data for its execution. We put Program_Identity_String along with the data to be locked so that no other program can unlock the locked blob of different protected software.

3.4.3 Security of Memory Access and Movement

The memory protection module prevents the data in the memory from being read or overwritten by malware with OS root privilege, as the attack will cause page fault and be trapped to the hypervisor. Then, the memory protection module will detect and deny the access. Assuming the security of the underlying hypervisor, it is infeasible for any OS components to modify the protected page without the verification from the hypervisor. After the unlocking operation from the secure vault, the hypervisor puts the decrypted data to the protected memory region of the program. Thus, other software programs cannot snoop or intercept the data.

One theme of the proposed protection mechanism is to isolate the protected code/data pages of a process by separating the page tables for the protected and the unprotected code, which has the capability of protecting the code and data from malicious software. It is worth mentioning that the modern commodity OS is a very

complex system with tens of millions of lines of code. For the security analysis purpose, we list several representative cases to illustrate the subtleties of the proposed hardware-based protection mechanism.

Processor Registers: When a task is scheduled out, there might be some sensitive information remnant in the CPU registers that might be readable by another process being scheduled in. In order to protect against this attack, the hypervisor clears the process specific registers and temporarily stores them in a hash table with the index key being the CR3 of the scheduled-out process. When the process is scheduled in, the register values are restored.

Shared data structures: The protected code will have to share the system stack with the unprotected code. Protecting the kernel stack is an ongoing task for our researches. In addition, the protected code will be writing to the buffers being used by other kernel components for kernel services or shared libraries. One way of mitigation is to let the shared library be marked as protected, as the shared library operates on the data pages allocated by the application. If the shared library allocates data pages and copies secret data to these data pages, the shared library marks these data pages as the protected.

DMA capable devices: DMA capable devices can directly read the memory without CPU intervention. Thus, it could bypass the hypervisor protections. These attacks can be largely addressed by the emerging device I/O virtualization technology.

Interrupts: Interrupts received by the CPU may change the flow of protected code by interleaving interrupt service routines (ISR) with the protected code. In our implementation, the interrupt handlers are not protected. As ISRs do not run in protected contexts, they do not have access to protected data. Every interrupt (received during the execution of protected code) gives up the CPU to unprotected code. Then, it transits back to the protected code. On every outgoing transition (protected--->unprotected), the hypervisor stores the EIP and the EBP registers. On the incoming transition (unprotected--->protected), the hypervisor validates the protected code EIP with the EIP previously cached.

Buffer Overflow attacks: The proposed mechanism does not address buffer overflow attacks. If there is buffer overflow attack vulnerability in the protected code, it can potentially be compromised and the secret will be stolen.

Software Unexpected Exits: When the software program has unexpected exits (e.g. program crashes) or the OS is to reclaim the memory, the security threat is that those secret data may still be in those memory regions. Then other high-privilege system software in the OS can access the memory regions. Thus, the mitigation is for the hypervisor to clear the hidden memory pages for the protected program in the event of premature program failure or OS reclamation of protected memory regions

Page File Protection: When the memory pages are swapped out to the disk, an attacker can modify the pages on the disk. This attack can be handled in two ways. First, the pages are locked in memory so that they do not get swapped out. The problem with this approach is that it is an approach that wastes memory. The second approach is that each protected page is hashed by the page swapper before the page is swapped out. The hash of the page is stored in protected memory. When the page is swapped in, the page swapper compares the hashes and raises an alert on a mismatch.

Debuggers: In this hypervisor based approach, the use of any debugger triggers a VM Exit which transfers execution control to the hypervisor by means of the DR7 register setting. Hence no other program including the OS will be able to view or modify the protected program's memory. This mitigation ensures that the protected program and its data are secure even when a debugger is used. It should be noted that the debugger still has the ability to modify memory regions belonging to other non-protected programs in the presence of protected programs.

4. IMPLEMENTATION AND EXPERIMENTAL RESULTS

For the performance evaluation of the proposed technology, we used a test system based on the Intel® i945GM chipset with an integrated gigabit Ethernet Network Interface Controller (NIC). The system had 1GB of main memory, and was powered by an Intel® Core™ Solo T1400 processor, which runs at 2 GHz and supports the virtualization technology of VT-x. We used an in-house developed lightweight hypervisor. Our hypervisor leverages VT-x to implement the memory protection module by using a variant of Virtual Translation Lookaside Buffer (VTLB) algorithm. The details of the algorithm are beyond the scope of this paper.

We have implemented three hyper calls--trap control to the hypervisor and use the services of the hypervisor. The protected software program registers with the hypervisor through the registration hyper call. Then other two hyper calls handle the cases of encryption and decryption of hypervisor, respectively.

After validating the request of the encryption hyper call, the hypervisor uses the AES-GCM module with the hypervisor 128-bit crypto key to encrypt the plaintext data. The encrypted blob is then placed into the memory pointed by the memory pointer sent in the encryption request. The application could store this blob on the local disk or at any un-trusted network location. On the decryption hyper call, the hypervisor compares a string agent identifier with the one cached at registration time that has been verified by the integrity measurement module. If the identifiers match, the hypervisor invokes the AES-GCM module to decrypt the encrypted blob and put the decrypted plaintext onto the protected memory of the software program.

We protect the code sections for encryption and decryption hyper calls (~100 lines of C code each in a caller program). In addition, a large data buffer (~1M bytes) aligned to the page boundary is protected. After the integrity measurement, the hypervisor puts the protection boundaries around the "start" and "finish" address sent in the registration call, which is the memory region to be protected.

Table I lists the timing results of those three hyper calls. To further break it down, the registration hyper call consists of verifying the memory against manifest (memory reading, crypto hash operation, and crypto digital signature) and the miscellaneous codes for context switch to the hypervisor. The total timing cost on the registration is about 227 μsec on the test platform. This happens only once during the program execution (the page swap case can be handled by the aforementioned optimization mechanism). Thus, we believe the performance penalty by the registration hyper call is graceful.

Table I: Cost of Hypervisor based Secure Locker on the test platform

Hypercall	Timing cost (μsec)
Registration	227
Encryption (on 1024-byte buffer)	22
Decryption (on 1024-byte buffer)	23

During the locking/unlocking service, the secure vault module in the hypervisor performs symmetric crypto operations. We choose the authenticated encryption method of AES-GCM as the default approach due to its good performance. In our measurement, it takes about 33 cycles/byte for the buffer of 1024-byte, i.e. 16.5 μsec in the test platform. In addition, the hyper call and the associated software glue costs about 10,000 CPU cycles, which is 5 μsec on the test machine.

5. CONCLUSIONS

Utilizing hardware virtualization techniques, we propose the architecture of protected page tables controlled by a hypervisor to protect sensitive data of software programs at runtime memory from malware with OS root privileges. In addition, we propose the secure vault mechanism in the hypervisor for locking/unlocking data exclusively for use by the associated validated programs. We believe the proposed techniques can complement the existing techniques by utilizing the underlying hardware virtualization technologies to protect program data in persistent storage and extending it to dynamic memory such that the data can be accessed only by the authenticated executing program. Finally, we demonstrate that such protections can be implemented for commodity operating systems with minimal impact to measurable system performance.

ACKNOWLEDGEMENT

We thank our former colleague, Uri Blumenthal, for his contribution in the architecture design of the secure vault module. We also thank our coworkers, Ravi Sahita and Uday Savagaonkar, for the results of memory protection based on hardware assisted x86 virtualization technologies.

REFERENCE

[1] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: a virtual machine-based platform for trusted computing," in *Proc. 2003 ACM SOSP*, pp. 193-206.

[2] R. S. Cox, J. G. Hansen, S. D. Gribble, and H. M. Levy, "A safety-oriented platform for web applications," in *Proc. 2006 IEEE Symposium Security and Privacy*, pp. 350-364.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proc. 2003 ACM SOSP*, pp. 164-177.

[4] R. Sailer, T. Jaeger, E. Valdez, R. Caceres, R. Perez, S. Berger, J. L. Griffin, and L. van Doorn, "Building a MAC-based security architecture for the Xen open-source hypervisor," in *Proc. 21st Annual Computer Security Applications Conf.*, 2005, pp. 276-285.

[5] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proc. 10th ACM Conf. Computer and Communications Security*, 2003, pp. 290-299.

[6] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static disassembly of obfuscated binaries," in *Proc. 13th USENIX Security Symposium*, 2004, pp. 255-270.

[7] D. Grawrock, *The Intel Safer Computing Initiative*, Intel Press, Jan. 2006.

[8] Trusted Computing Group, "TPM Main Part 3 Commands," *Specification Version 1.2, Level 2 Revision 94*, March 2006.

[9] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman, "A trusted open platform," *Computer*, July 2003, pp. 55-62.

[10] Microsoft Corporation, "BitLocker Drive Encryption: Technical Overview", Version 1.02, Apr. 2006.

[11] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot-a coprocessor-based kernel runtime integrity monitor," in *Proc. 13th USENIX Security Symposium*, pp. 179-194, 2004.

[12] A. Seshadri, A. Perrig, L. van Doorn, P. Khosla, "SWATT: software-based attestation for embedded devices," in *Proc. 2004 IEEE Symposium Security and Privacy*, pp. 272-282.

[13] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla, "Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems," in *Proc. ACM 2005 SOSP*, pp. 1-16.

[14] S. Bhatkar, R. Sekar, and D. C. DuVarney, "Efficient techniques for comprehensive protection from memory error exploits," in *Proc. 14th USENIX Security Symposium*, 2005,

[15] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proc. 11th ACM conf. Computer and Communications Security*, 2004, pp. 298-307.

[16] B. Chen, R. Morris, "Certifying program execution with secure processors," in *Proc 9th Workshop on Hot Topics in Operating Systems*, May 2003, pp. 133-138.

[17] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity operating systems," in *Proc. 2003 ACM SOSP*, pp. 207-222.

[18] Microsoft Corporation, "Microsoft Windows Vista Security Advancements," June 2006.

[19] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and implementation of a TCG-based integrity measurement architecture," in *Proc. 13th USENIX Security Symposium*, 2004, pp. 223-238.

[20] J. M. Agosta, J. Chandrashekar, D. H. Dash, M. Dave, D. Durham, H. Khosravi, H. Li, S. Purcell, S. Rungta, R. Sahita, U. Savagaonkar, E. M. Schooler, "Towards autonomic enterprise security: self-defending platforms, distributed detection, and adaptive feedback," *Intel Technology Journal*, vol. 10, no. 4, Nov. 2006.

[21] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," in *Proc. 2006 ACM ASPLOS*.

[22] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, L. Smith, "Intel virtualization technology," *Computer*, May 2005, pp. 48-56.

[23] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, J. R. Lorch, "Subvirt: implementing malware with virtual machines," in *Proc 2006 IEEE Symposium on Security and Privacy*.