Computing and Informatics, Vol. 32, 2013, 295–311

KERNEL CODE INTEGRITY PROTECTION BASED ON A VIRTUALIZED MEMORY ARCHITECTURE

Jianhua Sun, Hao Chen, Cheng Chang, Xingbang Li

School of Information Science and Engineering
Hunan University
410082 Changsha, China
e-mail: {jhsun, haochen, chengchang, xblee}@aimlab.org

Communicated by Jacek Kitowski

Abstract. Kernel rootkits pose significant challenges on defensive techniques as they run at the highest privilege level along with the protection systems. Modern architectural approaches such as the NX protection have been used in mitigating attacks, however determined attackers can still bypass these defenses with specifically crafted payloads. In this paper, we propose a virtualized Harvard memory architecture to address the kernel code integrity problem, which virtually separates the code fetch and data access on the kernel code to prevent kernel from code modifications. We have implemented the proposed mechanism in commodity operating system, and the experimental results show that our approach is effective and incurs very low overhead.

Keywords: Kernel rootkit, security, integrity protection, virtualization, Harvard architecture

1 INTRODUCTION

Operating system plays a critical role in modern computing systems. To support ever-growing range of hardware and provide more functionalities, operating system is becoming more and more complex steadily, which inevitably leads to increased security vulnerabilities.

We classify research related to mitigate the threat of kernel rootkits into two categories. The first are the mechanisms [10] that take advantage of hardware features (such as the privilege bits of segment table and page table, the NX-bit of

x86 architecture) to protect some important areas in kernel address space. Although these methods effectively protect the kernel to a certain degree, they have some obvious limitations when facing the threat of kernel rootkits. For example, in order to use NX bit in 32-bit x86 machines, one must set the PAE that is not supported on some legacy platforms. Sophisticated techniques [4, 16] used by determined attackers can still bypass these protection mechanisms.

In the second category, kernel code integrity checking [12, 13] is used to monitor and report violations of the kernel's control flow integrity by leveraging the kernel's CFG (control flow graph). These systems were designed to *detect* modifications to the kernel code. Thus they inherently are not capable of *preventing* the damages caused by kernel code execution.

In this paper, we propose a lightweight approach to protecting the kernel code of operating system. It implements the Harvard memory architecture, which separates the code from data in physical memory. After the operating system launches, all the illegal operations on the kernel code are redirected to somewhere in the physical memory that we call *shadow memory* so as not to cause any harmful impacts on the kernel code. This approach is implemented at the OS level and guarantees that all the malicious modifications to the kernel code do neither work nor hinder the OS from working normally and safely. As can be seen in the paper, our approach can protect the kernel code even the attacker gains the highest privilege, and at the same time, record the illegal attempts of modifying the kernel code, which is very useful for postmortem intrusion analysis. Our system makes use of some existing hardware features of x86 platform and only imposes very little overhead on system performance.

The rest of the paper is organized as follows. Section 2 states the threat model of this paper. Section 3 describes the details of system's design and implementation details. Evaluation results are presented in Section 4. Section 5 compares the system in this paper with SecVisor. Section 6 discusses the limitation and future work. Finally, Section 7 gives a brief description to related work and we conclude this paper in Section 8.

2 THREAT MODEL

The main characteristic of rootkits is stealthy. Rootkits often try to hide themselves on the compromised machine. Rootkit techniques have been developed along with the development of IDS and other defensive techniques. At the beginning, rootkits just replace or modify system files on the victim's hard disk. When researchers devised defensive methods such as file system integrity checkers [17], the rootkit makers then resort to new methods such as modifying static OS components or structures loaded in memory. The latest rootkits have begun to modify dynamically allocated OS objects.

As the size of OS kernel code steadily increases, more and more security vulnerabilities are introduced and the damage caused by these vulnerabilities is fatal.

The NIST National Vulnerability Database shows that in 2006, 81 and 31 security vulnerabilities were found in Linux and Windows XP, respectively.

Because the main purpose of most rootkits is not to crash a running system, generally they do not modify the kernel code directly. Instead, they usually inject malicious code into the kernel data area, and try to execute it as normal code. Rootkits have the capability to tamper with the kernel code due to the high privilege level they own. The Linux kernel do_brk vulnerability [11] enables attackers to change any page's privilege bits in the system, and to gain unrestricted access to almost any OS kernel code or data structures. Some attacker can even modify the physical address corresponding to a certain virtual address by manipulating the address translating mechanism [16]. If these vulnerabilities are exploited by rootkits to tamper with the kernel code, eventually they would lead to severe consequences to the operating system.

With the development of many defensive techniques, fewer attackers adopt the traditional way of code injection. Instead, more and more attackers inject malicious code into the code area or use the non-control-data attack technologies [6]. In this paper, we make the following assumption about the adversary model. First, the kernel rootkit has the highest privilege level in the victim machine. Second, in order to maintain stealthiness, the kernel rootkit needs to execute its malicious code in the kernel space.

3 DESIGN

3.1 Harvard and von Neumann Memory Architectures

Harvard memory architecture [1, 2] originates from the Harvard Mark I computer. The Harvard architecture has several intrinsic features that are different from von Neumann architecture [18]. First, instruction and data each have its own physical address space. In each address space, the co-existence of instruction and data is definitely not permitted. Second, instruction and data memory have separate hardware paths to the central processing unit (CPU). Since they are stored at separate physical memories, this feature allows instructions to be fetched and data to be accessed at the same time by using different system buses.

As we can see, the features that Harvard memory architecture provides enable a computer system to access the instruction and data in one clock cycle, which obviously improves the overall system performance. Harvard architecture is frequently used in DSPs, AVRs and ARMs, and can also be found in some operating systems like Vxworks [19].

Most modern computers implement a von Neumann memory architecture in which code and data share the same flat address space. The co-existence of code and data in one address space is the root cause of the code injection problem [15]. It seems that the attackers can always figure out a way to bypass the defending system to inject malicious code or modify the kernel code, despite of many advances in defensive techniques. The splitting memory model of Harvard architecture is

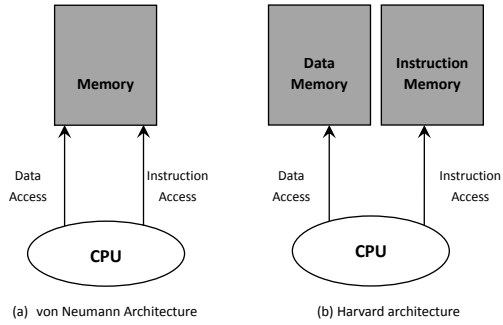


Fig. 1. Harvard architecture using separate memory and system bus to access data and instruction

a desirable feature to prevent kernel from code injection attacks at its root, because the data and instruction are stored in separate physical memories and accesses to these two memory regions are under hardware mediation. Thus instructions and data cannot be addressed to each other, which means that the code injection is impossible under Harvard memory architecture because code cannot be read/written like data and data cannot be executed like code, even at the highest privilege level.

3.2 Implementing Harvard Memory Architecture on x86 Platform

Harvard memory architecture can mitigate kernel code injection effectively, but modern commodity hardware such as x86 is not capable of providing this functionality due to its unified memory architecture. Although it is possible to make modifications to the existing architecture to support a splitting memory architecture, this would not be beneficial to legacy systems. Furthermore, modifying a widely deployed processor is not always practical.

Although most of the general purpose CPUs are designed based on von Neumann architecture, and use a single address space to store both data and instructions, another option for building the Harvard memory architecture is to take advantage of existing architectural features in commodity hardware. Leveraging the PageTable and TLBs on x86 systems, we can create a virtualized Harvard memory architecture in modern operating systems with only minor modifications to kernel source code. Without relying on a hardware implementation, the proposed system can run on conventional x86 hardware, and offer protection to legacy systems as well as newly developed systems.

4 IMPLEMENTATION

Our goal is to ensure the integrity of kernel code at a minimal cost (slight overhead on performance and minor modifications to kernel source code).

On x86 CPU, given a linear address *va*, the MMU (Memory Management Unit) translates it to a physical address, denoted as $Mapping\_d(va) = pa$.

This mapping of addresses is the same to instruction fetch and data access, because the translation process uses the same page table provided by the OS. The linear address *va* is always mapped to the physical address *pa*.

However, in our system, for $\forall va \in V$ (V is the linear address space of kernel code), we have:

$$\text{For instruction fetch at address } va : \text{Mapping\_}i(va) = pa_{\text{exec}} \tag{1}$$

$$\text{For data access at address } va : \text{Mapping\_}d(va) = pa_{\text{data}} \tag{2}$$

$pa_{exec} \in P_I$, $pa_{data} \in P_D$ and $P_I \bigcap P_D = \Phi$. P_I is the instruction physical address space, P_D is the data physical address space which we call *shadow memory*. All the suspicious operations performed on kernel code are redirected to this area.

Once this protection mechanism is established, it ensures the integrity of the kernel code as long as the operating system is running. The architecture of our system is represented in Figure 2.

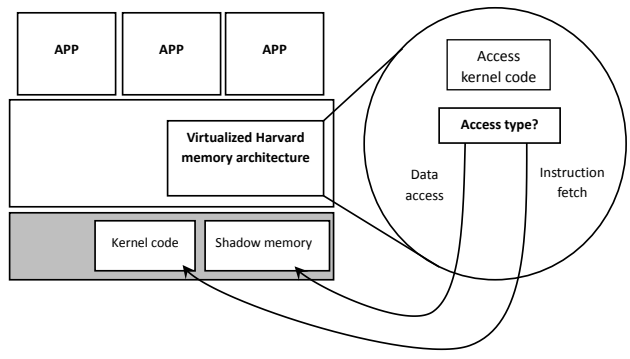


Fig. 2. System architecture

Based on Linux/x86, our system makes a few modifications to address translation component in Linux kernel and takes advantage of the separated instruction and data TLBs. Before giving detailed descriptions of our system, we first present a brief overview about the address translation mechanism in Linux and the separated TLB management in x86 systems.

Translation of virtual address to physical address in Linux/x86 CPU is accomplished by a multi-level translation scheme which uses several types of page tables. Here we only describe the last level translation (the translation from linear address to physical address) due to its direct relevance to our purpose.

Every time when CPU generates a linear address, MMU looks up the page table to perform the translation. Each entry in the page table is called a PTE (Page Table Entry). The page table resides in main memory, if the corresponding PTE is not

found, an additional memory access is required to translate the linear address at the cost of tens of hundreds of cycles. To eliminate this cost, translation lookaside buffers (TLB) are adopted to cache PTEs to speed up the address translation process. The steps to perform translation of linear address involving TLBs are as follows.

1) The CPU generates a virtual address.
2, 3) The MMU fetches the appropriate PTE from the TLB.
4) The MMU translate the linear address to the physical address and then send it to cache/main memory.
5) The cache/main memory sends the requested data back to the CPU.

If in step 2 there is a TLB miss, the MMU will look up the page table in main memory to find the corresponding physical address and then move to step 4 and cache the corresponding PTE in TLB by replacing an existing entry. As noted above, visiting the TLB is an indispensable step to translate a virtual address and cannot be bypassed.

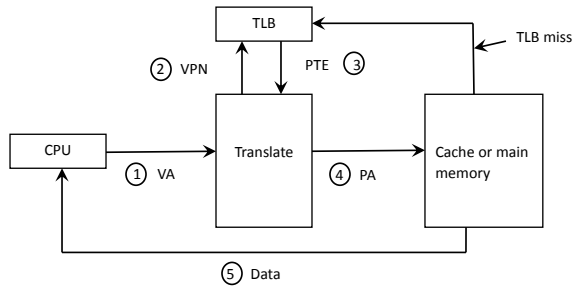


Fig. 3. Address translation with TLB

In Section 3.1 we mentioned that many general purpose CPUs have features such as separated instruction and data caches and TLBs, which can be leveraged to emulate the Harvard memory architecture at the software level.

Although modern CPUs are getting more and more complex, generally there are 5 pipeline stages needed to execute an instruction: *fetch*, *decode*, *execute*, *memory* and *write back*. For the purpose of high concurrency, the five stages should be interleaved with each other. Thus the separated caches (I-Cache and D-Cache) and TLBs (I-TLB and D-TLB) have been introduced to ensure the two stages of *fetch* and *memory* would not interfere with each other. While CPU accesses memory in the fetch stage, it looks for corresponding physical address of the instruction page in I-TLB. Similarly, while CPU accesses memory in the memory stage, it looks for corresponding physical address of the data page at D-TLB, as shown in Figure 4.

Although theoretically the two corresponding PTEs (in I-TLB and D-TLB) of a specific linear address should be the same, there is actually no hardware mechanism to ensure this, which enables us to implement a virtualized Harvard memory

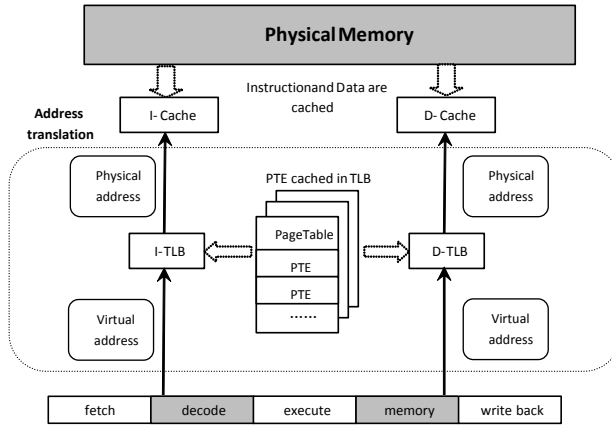


Fig. 4. Address translation at different pipeline stage

architecture. We can enhance the operating system to provide two different page tables for the fetch stage and memory stage respectively, but actually we do not have to maintain two page tables by means of keeping different PTEs in different TLBs. By doing so, we can guarantee that any modifications to the kernel do not influence the normal execution of kernel code. In the following, we elaborate on our implementations.

1. During system initialization, we allocate a continuous memory region whose size is the same as that of the kernel code. This region is called *shadow memory*, and is pointed to by a pointer variable kernel_text_mirror(unsigned long *const kernel_text_mirror). We store a copy of the kernel code in this area.

   There is a one-to-one mapping between the frames of shadow memory and the frames of kernel code. The first frame number of the kernel code is text_start and the first frame number of the shadow memory is mirror_start. Then a kernel code frame ppn's corresponding frame number in shadow memory is shadow(ppn) which can be calculated by the following formula:

$$\text{shadow}(ppn) = ppn + (\text{mirror\_start} - \text{text\_start}) \tag{3}$$

2. Change the ppn bits of all the kernel code page's PTEs to shadow(ppn) temporarily, then load them to D-TLB (reading one byte of a particular page can cause the system to load the corresponding PTE to D-TLB).
3. Recover all the PTE's ppn bits that have been changed, and set the R/W bit to 0.
4. Modify the page fault handler: if the page fault is caused by trying to tamper with the kernel code, change the corresponding PTE's R/W bit to 1, then go to step 2 and 3.

With our modification the address translation of kernel code pages is as shown in Figure 5.

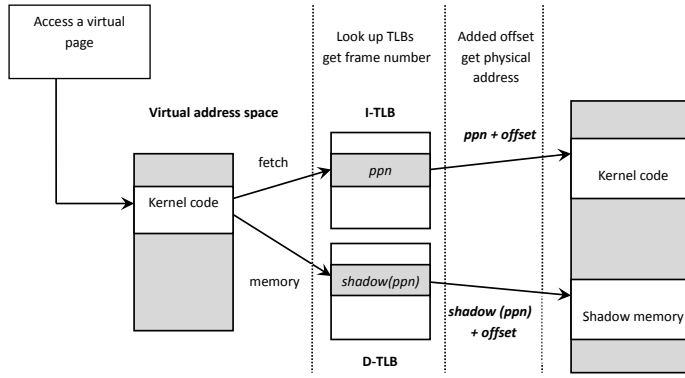


Fig. 5. Address translation of kernel code pages after our modification

After system initialization, this double-mapping address translation of kernel code pages is established. Generally speaking, the PTEs of kernel code pages are resident in TLB, because their Global bits are set. Even if they are flushed out accidentally, since the kernel code pages are set as Read-Only, page fault will be generated while these pages are accessed and the double-mapping address translation mechanism will be established again at step 4 in the page fault handler.

4.1 Effectiveness and Impact on Operating System

When the protection mechanism is established, querying the TLB is the first step in address translation, so all the write operations are directed to the shadow memory region. Even if the PTEs corresponding to kernel code pages are flushed out off TLB, in the page fault handler invoked by writing to kernel code page, the appropriate PTE modified will be loaded into D-TLB again and the defense system will be re-established. Therefore, the protection mechanism is always on in the system, and any malicious operations attempting to make damages to the kernel code can not manage to bypass it. In addition, all the malicious operations return without any explicit response indicating any failures in attack operations, making the attacker unaware of the unsuccessful attack attempts.

All the data accesses have been redirected to another region in memory. Although all the write operations on kernel code should be regarded as malicious behavior, the read operations on kernel code are considered as legal. However, if the access to kernel code is in the memory stage instead of the fetch stage, this access should be regarded as suspicious because its intension is not to execute the kernel code but to read the kernel code for certain other reasons. This perspective is sound because reading kernel code is definitely not an essential operation for the normal

execution of operating system. In our current implementation, we do not consider security tools performing code intergrity checks, which may need to read kernel code pages. Thus any data accesses to kernel code in the memory stage rather than the fetch stage is considered suspicious. As the legal operation (execution) on kernel code is achieved through I-TLB and D-TLB, manipulating the content of D-TLB for our purpose will not cause any unacceptable consequences on the operating system's normal functionalities.

Our system makes full use of the TLB, all the modified PTEs are loaded to the TLB and are seldom flushed. If it is the case, the re-loading procedure will finish in the page fault handler. For these reasons, the defense system will not make any substantial overhead on performance.

When the kernel code page's PTEs are loaded into D-TLB and kept there, the capacity of D-TLB is actually reduced, which would lead to negative impact on system performance.

In addition, our approach makes minor changes to the source code of page table initialization and page fault handler. All the code added is not more than 200 lines and have been debugged and analyzed carefully (all the global variables such as kernel_text_mirror, text_start and mirror_start are set as Read-Only) in order not to introduce new vulnerabilities into the kernel.

5 EXPERIMENTS

5.1 Performance Evaluation

Our experimental platform was a PC with one Pentium 4 3.0 G processer and 512 MB meomory. In our first experiment the operating system was Fedora Core 4 with a Vanilla Linux 2.6.11 kernel (presented as Linux in the figures), the second experiment was conducted on Fedora Core 4 with a modified kernel (presented as Linux_M in the figures).

Benchmark	Original (μs)	Modified (μs)	Overhead
Arithmetic Test	206.6	203.8	1.36 %
Dhrystone 2	346.5	332.7	3.98 %
File Copy 1 024 bufsize 2 000 maxblocks	833.7	827.6	0.73 %
File Copy 256 bufsize 500 maxblocks	593.1	586.3	1.15 %
File Copy 4 096 bufsize 8 000 maxblocks	1 187.5	1 164.5	1.94 %
Pipe Throughput	411.0	419.8	−2.14 %
Process Creation	998.5	963.1	3.55 %
Shell Scripts	852.8	836.7	1.89 %
System Call Overhead	263.1	261.9	0.46 %

Table 1. Unixbench results

We used the lmbench benchmark to measure the overhead of different kernel operations. The experimental results are shown in Figures 6, 7 and 8. In order to

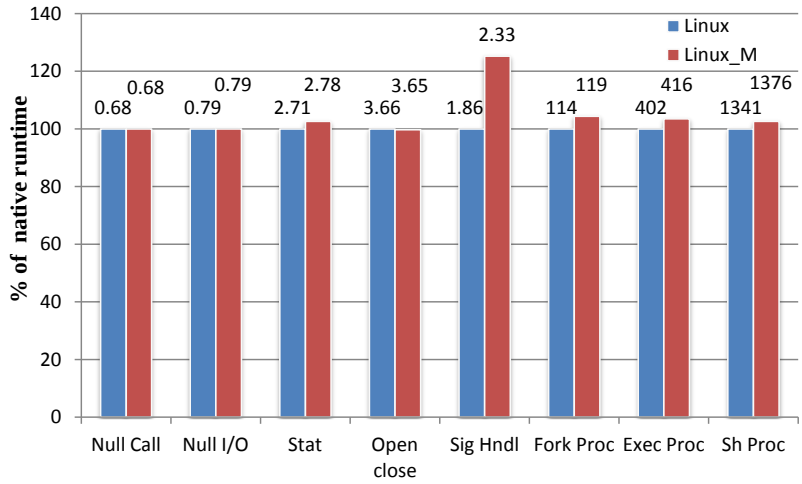


Fig. 6. Execution times of lmbench process and memory microbenchmarks. All times are in μs.

show the performance overhead when the protection mechanism is enabled, The Y axis shows the normalized execution time to native Linux (lower is better). The figures above the histogram represent real test results. In addition, we used the micro-benchmark suite UnixBench. The experimental results are shown in Table 1, where larger value means better performance.

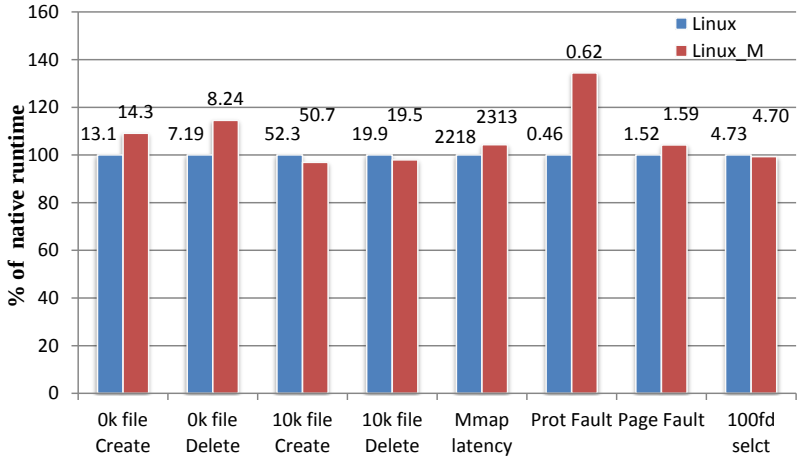


Fig. 7. Execution times of lmbench file and VM latencies microbenchmarks. All times are in μs.

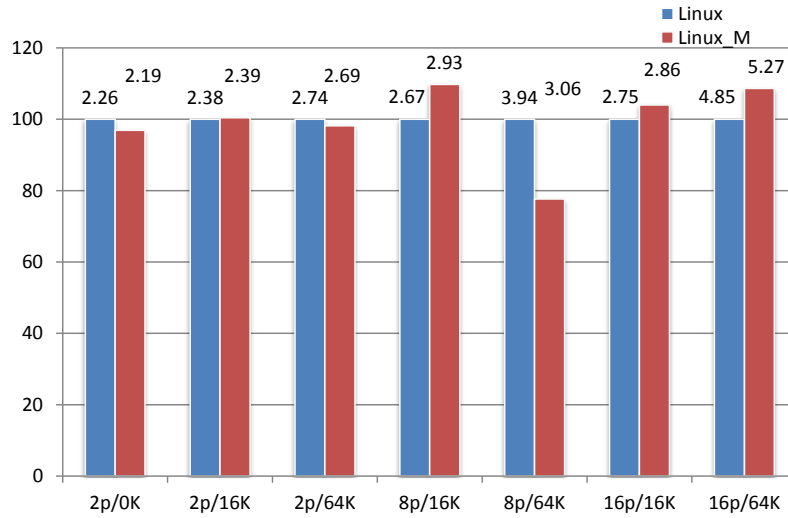


Fig. 8. Execution times of lmbench context switch microbenchmarks. All times are in μs.

It can be observed from the experimental results of lmbench that there is no significant slowdow in runtime when the protection mechanism is enabled. The maximum overhead is 34 % in the case of Prot Fault in Figure 7, which measures the time taken by the kernel to handle a write access violation. The overheads in other cases are all less than 10 %. From Table 1, we can see that the overheads are all less than 3.98 %.

In addition, the effect of putting kernel code PTEs in D-TLB is minimal. The number of entries of TLB on our testing machine is 128, the kernel code page's PTEs only occupied a small fraction of the D-TLB cache and thus it did not incur any visible performance overhead.

5.2 Validation Test

The threat model of this paper is the kernel rootkit trying to tamper with kernel code. We implemented a new rootkit by modifying an existing one to evaluate the effectiveness of our system.

The adore-ng 0.56 is a kernel rootkit on Linux 2.6. It is a Linux loadable kernel module. By modifying the entry address it hijacks some system calls of Linux kernel. The modified adore-ng 0.56 runs on Linux 2.6.11. It directly tampers with the code of system call functions in kernel code and changes the content of the functions' page to '0'. Obviously, this will cause the entire system to break down immediately. In our experiment, we installed the modified adore-ng 0.56 on the Fedora Core 4 with Vanilla Linux 2.6.11. Without our protection mechanism, the system crashed right after the installation of the rootkit, indicating that the kernel

code had been modified so that it can not run normally. On the same machine with our system switched on, the rootkit's installation still succeeded, however the system worked as usual. We obtained the address of the variable kernel_text_mirror from the file "System.map", which is the starting address of shadow memory, and then printed out the content of shadow memory through /dev/mm. The content of the shadow memory was all '0's, which indicates that the malicious attempt to tamper with kernel code has been directed to the region of shadow memory successfully.

We also extended some other rookits with the ability to tamper with kernel code in the same way as described above. When evaluating the effectiveness of our system, it is important to note that the evaluation criteria is whether it can prevent these rookits from modifying the kernel code, but not prevent them from installing. Table 2 lists the experimental results. In fact, all these rootkits can be installed normally, but they can not achieve the intended malicious goals.

Rootkit	Attacks	Whether it can prevent modifying the kernel code
Adore-ng 0.56	LKM	yes
eNyeLMV1.2	LKM	yes
override	LKM	yes
Phalanx b6	/dev/mem	yes

Table 2. Validation test

6 COMPARING WITH SECVISOR

6.1 Performance Comparison

Although the test environment of SecVisor is different from the one used in this paper, we present the comparison of these two systems by examining the relative performance loss.

From Tables 2 and 3, we can see that SecVisor causes significant overheads in all testing scenarios compared with the native Linux. The Null Call shows the overhead of a round trip between user and kernel. In this case, SecVisor slows down the system call by 256x. In other cases, we still see 15-110x slowdowns. The slowdown caused by SecVisor can be attributed to two main reasons: (1) SecVisor needs to maintain data structures such as shadow page table and so on due to the introduction of the VMM layer. (2) The time spent on address translation, page faults handling and context switching increases substantially.

Benchmark	Null Call (μs)	Fork (μs)	Exec (μs)	Prot Fault (μs)	PF (μs)
Linux	0.10	139	410	0.248	1.71
SecVisor	25.6	2 274	6 203	27.3	35.1

Table 3. Execution times of lmbench file and VM latencies Microbenchmarks in SecVisor

Benchmark	2p/0k (μs)	2p/16k (μs)	2p/64k (μs)	8p/16k (μs)	8p/64k (μs)
Linux	0.56	0.64	3.19	1.48	12.9
SecVisor	54.3	52.7	53.6	63.3	75.8

Table 4. Execution times of lmbench context switch microbenchmarks in SecVisor

6.2 Functionality Comparison

Table 5 lists the comparison of functionality between SecVisor and our system. It is obvious that SecVisor provides more protection than our system, such as preventing DMA attacks and kernel data attacks; but our system is more simple and easier to deploy as shown in the last row of Table 5, and is applicable to legacy systems without hardware virutalization that is required by SecVisor. Futhermore, our system also has the ability of recording malicious behaviors, which is not available to SecVisor.

Functionality, Usability and Strategy	Our system	SecVisor
Protect the kernel code page	Yes	Yes
Protect the mixed data code page	Yes	No
Prevent attacking the kernel code by using LKM	Yes	Yes
Prevent attacking the kernel code by using /dev/mem	Yes	Yes
Prevent attacking the kernel code by using DMA	No	Yes
Prevent the kernel data injection attacks	No	Yes
Record the attacker behaviors	Yes	No
Strategies to combat attacking	Relocating Diverting Installing system patches	Defense Preventing Modifying Linux kernel
Depend on hardware features	Applicable to legacy systems	Needs the support of hardware virtualization
The methods of installation	Recompiling the kernel Starting up from the new kernel	Recompiling the kernel Installing the kernel module Adjusting the system boot sequence

Table 5. Comparison of the function of SecVisor with the system in this paper

7 LIMITATION AND FUTURE WORK

7.1 Security of the Whole Kernel Space

Our system is designed to deal with the challenges in kernel code modifications rather than to solve the problem of existing threats which inject code into the kernel data space. In fact, our approach can be applied to these existing threats by extending the virtualized Harvard memory architecture to the entire kernel space. A virtualized Harvard memory architecture in which code and data are totally separated can significantly enhance the kernel security.

7.2 Further Usage of the Shadow Memory

Although the suspicious operations are directed to a safe memory region, we do not make a full use of the information recorded in this region. The information recorded by our protection mechanism is very useful for postmortem intrusion analysis. There are many security tools trying to record all the operations of the OS over a period of time so that the intrusion analysis and intrusion recovery are able to work. BackTracer [5] and Taser [8] log all the information needed using system call interposition. Later, when the system is found intruded, the information logged in a safe place is used to replay the system's execution from a certain point. By doing this, analysts can observe how the intrusion is conducted, and which components are tainted by this intrusion, and then recover the entire system to a safe state. ReVirt [7] is a logging and recovery system based on VMM. Logging the operations of OS at instruction-level, combined with the advantage of strong isolation provided by VMM, ReVirt has much better reliability than many OS-level loggers. One disadvantage of these system loggers is that they simply log and replay all the operations executed without judging whether the operations are illegal or interfering with their execution. If the approach in this paper is to be used in the logging system, all the suspicious operations are recorded without any harmful impact on the system. The logs recorded are very useful for analyzing the behavior of malicious programs.

8 RELATED WORK

Ryan Riley [15] proposed a method virtually splitting memory to prevent code injection attack at user-level. The approach is similar to ours; but to manipulate the content in TLBs, the single step execution mode was frequently used, which would significantly affect the speed of instruction execution. The intensive context switches of user processes also cause large overhead. These weaknesses make Ryan Riley's approach unpractical in performance-critical systems.

Focusing on the different characteristics of the two kinds of memory access, Wurster et al. [20] implemented two address translation mechanisms to bypass self

check-summing. For the same purpose, Nathan E. Rosenblum [14] developed an extension to the Xen hypervisor to implement context sensitive paging mapping. The methods they used are similar: after modifying the target program, they turn on the mechanism of context sensitive mapping so that when the self check-summing code is checking the integrity of specified program, what it reads is actually other physical pages holding the target program's unchanged code. But the code the program executes will be the modified version. In this way, the self check-summing process is successfully bypassed.

Sparks and Butler [16] proposed a new rootkit: Shadow Walker. It adopts a method similar to our approach. By modifying Windows XP's page fault handler, it presents a fake memory space to hide its malicious code in order to circumvent memory scanners.

9 CONCLUSION

In this paper, a virtualized Harvard memory architecture was proposed and a proof-of-concept prototype was implemented in Linux operating system, which separates the code fetch and data operation on the kernel code. The effectiveness of our system to resist kernel level attacks was evaluated and the impact on operating system performance was also analyzed in detail. The experiment results show that our approach is effective and practical.

Acknowledgments

The authors are grateful to the anonymous reviewers for their helpful feedback. This research was supported in part by the National Natural Science Foundation of China under grants 61272190, 61173166 and 60803130, the Program for New Century Excellent Talents in University, and the Fundamental Research Funds for the Central Universities of China.

REFERENCES

[1] Aiken, H. H.: Proposed Automatic Calculating Machine. 1937, reprinted in The Origins of Digital Computers Selected Papers, Second Edition, 1975, pp. 191–198.

[2] Aiken, H. H.—Hopper, G. M.: The Automatic Sequence Controlled Calculator. 1946, reprinted in The Origins of Digital Computers Selected Papers, Second Edition, 1975, pp. 199–218.

[3] Barham, P.—Dragovic, B.—Fraser, K.—Hand, S.—Harris, T.—Ho, A.—Neugebauer, R.—Pratt, I.—Warfield, A.: Xen and the Art of Virtualization. In Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003), ACM Press, pp. 164–177.

[4] Buffer Overflow Attacks Bypassing DEP (NX/XD bits) – Part 2: Code Injection. Available on: http://www.mastropaolo.com/2005/06/

05/buffer-overflow-attacks-bypassing-dep-nxxd-bits-part-2-code-injection/.

[5] Chen, P. M.—King, S. T.: Backtracking Intrusions. In Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003), Bolton Landing, NY USA, pp. 223–236.

[6] Chen, S.—Xu, J.—Sezer, E.—Gauriar, P.—Iyer, R.: Non-Control-Data Attacks are Realistic Threats. In Proceedings of the Usenix Security Symposium 2005, pp. 177–192.

[7] Dunlap, G. W.—King, S. T.—Cinar, S.—Basrai, M.—Chen, P. M.: ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI) 2002, pp. 211–224.

[8] Goel, A.—Po, K.—Farhadi, K.—Li, Z.—de Lara, E.: The Taser Intrusion Recovery System. In Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP 2005), pp. 163–176.

[9] Kivity, A.—Kamay, Y.—Laor, D.: KVM: The Linux Virtual Machine Monitor. Linux Symposium 2007.

[10] Securing Memory. Available on: http://www.kernelthread.com/publications/security/smemory.html.

[11] Linux Kernel do brk() Vulnerability. Available on: http://www.isec.pl/papers/linux_kernel_do_brk.pdf.

[12] Loscocco, P. A.—Wilson, P. W.—Pendergrass, J. A.—McDonnell, C. D.: Linux Kernel Integrity Measurement Using Contextual Inspection. In Proceedings of the 2007 ACM Workshop on Scalable Trusted Computing, pp. 21–29.

[13] Patil, S.—Kashyap, A.—Sivathanu, G.—Zadok, E.: 3FS: An In-Kernel Integrity Checker and Intrusion Detection File System. In Proceedings of LISA '04: Eighteenth Systems Administration Conference, pp. 67–78.

[14] Rosenblum, N. E.—Cooksey, G.—Miller, B. P.: Virtual Machine-Provided Context Sensitive Page Mappings. In Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '08), pp. 81–90.

[15] Riley, R.—Jiang, X.—Xu, D.: An Architectural Approach to Preventing Code Injection Attacks. In Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2007), pp. 30–40.

[16] Sparks, S.—Butler, J.: ShadowWalker: Raising the Bar for Windows Rootkit Detection. In Black Hat Japan, Tokyo, Japan 2005.

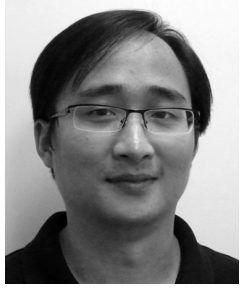
[17] Tripwire Inc. Available on: http://www.tripwire.com/.

[18] von Neumann, J.: First Draft of a Report on the Edvac. 1945, Reprinted in the Origins of Digital Computers Selected Papers, Second Edition, 1975, pp. 355–364.

[19] Wind river Vxworks. Available on: http://www.windriver.com/vxworks/.

[20] Wurster, G.—van Oorschot, P.—Somayaji, A.: A Generic Attack on Checksumming-Based Software Tamper Resistance. In Proceeding of 2005 IEEE Symp. Security and Privacy, 2005, pp. 127–138.

Jianhua Sun is an Associate Professor at the School of Information Science and Engineering, Hunan University, China. She received the Ph. D. degree in computer science from Huazhong University of Science and Technology, China in 2005. Her research interests include security and operating systems.



Hao Chen received the B. Sc. degree in chemical engineering from Sichuan University, China, in 1998, and the Ph. D. degree in computer science from Huazhong University of Science and Technology, China in 2005. He is now an Associate Professor at the School of Information Science and Engineering, Hunan University, China. His current research interests include parallel and distributed computing, operating systems, cloud computing and systems security. He published more than 40 papers in top journals such as the IEEE Transactions on Parallel and Distributed Systems (TPDS) and IEEE Transactions on Computers (TC), and in renowned conferences like IPDPS, IWQoS, HiPC, and CCGrid. He is a member of the IEEE and the ACM.

Cheng Chang is a Ph. D. student at the School of Information Science and Engineering, Hunan University, China. His research interests include cloud computing, parallel and distributed computing and security.

Xingbang Li was a former graduate student at the School of Information Science and Engineering, Hunan University, Chian. His research interests include operating systems and security.