LETTER *Special Section on Security, Privacy and Anonymity in Computation, Communication and Storage Systems*

Kernel Rootkits Detection Method by Monitoring Branches Using Hardware Features*

Toshihiro YAMAUCHI[†a)], ***Member and*** **Yohei AKAO**[†], ***Nonmember***

SUMMARY An operating system is an essential piece of software that manages hardware and software resources. Thus, attacks on an operating system kernel using kernel rootkits pose a particularly serious threat. Detecting an attack is difficult when the operating system kernel is infected with a kernel rootkit. For this reason, handling an attack will be delayed causing an increase in the amount of damage done to a computer system. In this paper, we propose Kernel Rootkits Guard (KRGuard), which is a new method to detect kernel rootkits that monitors branch records in the kernel space. Since many kernel rootkits make branches that differ from the usual branches in the kernel space, KRGuard can detect these differences by using the hardware features of commodity processors. Our evaluation shows that KRGuard can detect kernel rootkits that involve new branches in the system call handler processing with small overhead.

key words: *kernel rootkit detection, last branch record, operating system, system security*

1. Introduction

Rootkits are programs that hide malicious behaviors from computer users. There are two types of rootkits: user rootkits that run at the user level and kernel rootkits that run at the kernel level. In particular, attacks of kernel rootkits against an operating system (OS) pose a serious threat. Kernel rootkits modify the OS kernel and rewrite the data outputted by the OS. Therefore, detecting methods based on the output data of the OS are ineffective. For example, anti-virus software running at the user level cannot detect kernel rootkits. Thus, detecting kernel rootkits is difficult, and various methods to detect them have been proposed.

We mentioned that the existing kernel rootkit detection methods cannot resolve all of the following problems simultaneously [3]: (1) cannot detect kernel rootkits immediately, (2) cannot keep the extensibility of the OS kernel, and (3) cannot be applied to different OS and OS versions. To resolve those problems, we proposed a method to detect kernel rootkits by checking the kernel stack [3]. However, this method (4) cannot detect kernel rootkits that use instructions that do not push data into the kernel stack (e.g., the *jmp* instruction). In addition, a method proposed in [4] requires a virtual machine monitor to detect kernel rootkits in a guest virtual machine.

Manuscript received January 16, 2017.
Manuscript revised May 8, 2017.
Manuscript publicized July 21, 2017.

[†]The authors are with the Graduate School of Natural Science and Technology, Okayama University, Okayama-shi, 700–8530 Japan.

*Preliminary works of this paper were presented at [1], [2].

a) E-mail: yamauchi@cs.okayama-u.ac.jp

DOI: 10.1587/transinf.2016INL0003

In this paper, we propose Kernel Rootkits Guard (KRGuard), which is a new method to detect kernel rootkits and resolve problems (1)–(4) simultaneously. KRGuard detects kernel rootkits by monitoring the branch records in kernel space recorded by the hardware features of commodity processors. KRGuard utilizes the fact that many kernel rootkits make branches that differ from the usual branches. Therefore, KRGuard can detect kernel rootkits with new branches between the hook function before calling the system call handler, and the hook function before calling the system call service routine. In addition, we describe the limitations, the implementation of KRGuard as a kernel module on Linux, and the evaluation results of KRGuard.

The preliminary version of this paper appeared in [1], [2]. This paper describes the originality, effectiveness and limitations of KRGuard in detail.

2. Design of KRGuard

2.1 Concept

KRGuard utilizes the fact that many kernel rootkits make branches that differ from the usual branch path. Previous research [5] indicates that 96% of all kernel rootkits employ control-flow modifications, making branches different from the usual types. For example, Fig. 1 shows the change in the control-flow when the system call control-flow is modified by kernel rootkits. Usually, after invoking a system call, the control moves from the system call handler to each system call service routine. On the other hand, when a computer system is infected with kernel rootkits, the control moves from the system call handler to the malicious code prepared by the attacker before moving to each system call service routine. In the malicious code, the processing that hides attacks is executed. KRGuard detects kernel rootkits by monitoring branch records in kernel space and by detecting control-flow modifications. KRGuard uses the Last Branch Record, a recent feature of Intel processors for monitoring branch records in kernel space.

2.2 Last Branch Record

Last Branch Record (LBR) [6] is a recent feature of Intel processors that was introduced in the Nehalem architecture. When the LBR is enabled, the CPU records the address of a branch instruction and its target instruction (branch record) on the LBR stack register, which can store up to 16 entries.

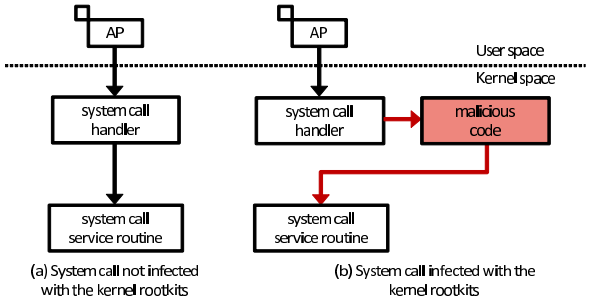

Fig. 1 Changes in the control-flow when system call control-flow is modified.

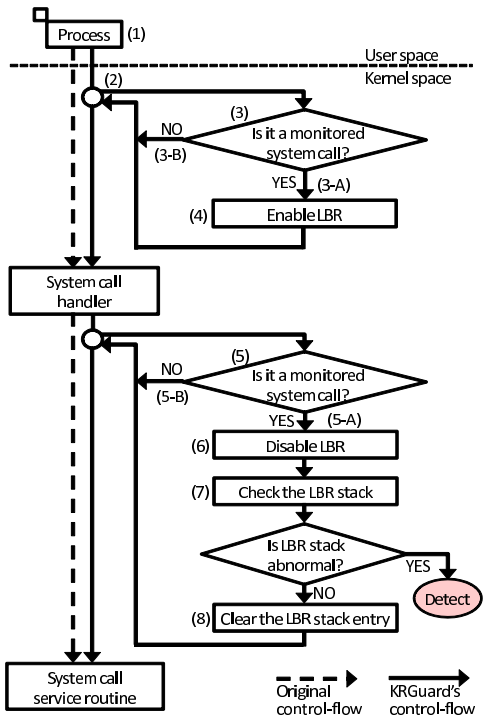

Fig. 2 Processing flow of KRGuard.

When more than 16 entries are recorded, the oldest stack data is overwritten. Monitoring branch records using the LBR has the following advantages:

(1) It can record all branch records in the kernel. Therefore, it can monitor branch records recorded by instructions that do not push data into the kernel stack.
(2) It is transparent to the OS structure.
(3) It generates minimal overhead [7].

2.3 Overview

KRGuard detects kernel rootkits that modify the control-flow of a system call by monitoring the branch records using the LBR in Linux. It should be noted that we do not address attacks on KRGuard itself in this paper.

Figure 2 shows the processing flow of KRGuard. KRGuard detects kernel rootkits that modify the control-flow of the system call as follows:

(1) A user program invokes a system call.
(2) KRGuard hooks the transition to the system call handler.
(3) KRGuard judges whether the invoked system call is a monitored system call, and the following processing is executed:
(A) If the invoked system call is a monitored system call, then control is given to Step (4). (B) Otherwise, KRGuard does nothing, and control is given to the system call handler.
(4) The LBR is enabled (to start monitoring branches), and control is given to the system call handler.
(5) The following processing is executed:
(A) If the invoked system call is a monitored system call, KRGuard hooks the transition to each system call service routine, and control is given to Step (6). (B) Otherwise, control is given to each system call service routine.
(6) The LBR is disabled (to stop monitoring branches).
(7) KRGuard checks branch records in the LBR stack. If branch records in the LBR stack are abnormal (see Case (2) described in Sect. 2.4), KRGuard alerts the user.
(8) Branch records in the LBR stack are cleared, and control is given to each system call service routine.

KRGuard monitors the following 13 system calls that are likely to be modified by attackers: exit(), fork(), read(), write(), open(), close(), execve(), ioctl(), readlink(), stat64(), lstat64(), getuid32(), and getdents64(). Those system calls monitored by KRGuard are determined by referring to [3], [8], and [9].

Using these steps, KRGuard monitors the branch records between the invoking system call and the transition to each system call service routine.

2.4 Checking Branch Records in the LBR Stack

KRGuard detects kernel rootkits based on the quantity of branch records in the LBR stack.

In KRGuard, branch records recorded by the LBR are classified in the following four ways:

(1) When the computer system is not infected with kernel rootkits, the LBR records two pieces of branch records.
(2) When the computer system is infected with kernel rootkits, the LBR records more than two pieces of branch records by processing the kernel rootkits.
(3) When the process is traced (by the feature of ptrace), the LBR records more than two pieces of branch records by processing the trace.
(4) When an interrupt occurs, the LBR records more than two pieces of branch records by processing the interrupt.

When the quantity of branch records contained in the LBR stack is two, KRGuard determines that the computer system is not infected with kernel rootkits. When the quantity is greater than two, KRGuard verifies whether the pro-

cess is traced intentionally by the user by outputting process information to the user. If the user intentionally traces the process, the user can confirm that the process is intentionally traced by checking the process information outputted by KRGuard. When the process is not intentionally traced, KRGuard determines that the computer system is infected with kernel rootkits. Handling a case in which an interruption occurs is an issue that we will consider in the future.

2.5 Detectable Kernel Rootkits and Limitations

KRGuard monitors the branches between the hook function before calling the system call handler, and the hook function before calling the system call service routine. Therefore, KRGuard can detect kernel rootkits with new branches within this range. For this reason, KRGuard has the advantage of being able to newly detect kernel rootkits that embed a branch instruction in a system call handler, compared with the method of Ikegami et al. [3].

However, in the kernel rootkits that tamper with the system call handler, it is not possible to detect kernel rootkits that embed a branch instruction in the codes of the system call service routine. In addition, some kernel rootkits do not call legitimate system call service routines. In this case, because the legitimate system call service routines are not hooked and a branch information comparison is not performed, they cannot be detected by KRGuard.

However, by introducing a mechanism for checking whether the corresponding system call service routine was called in the previous system call when a system call was issued, as in the method of Ikegami et al. [3], we can detect the kernel rootkits that do not call legitimate system call service routines. In addition, kernel rootkits that cannot be detected by the KRGuard can be dealt with by using methods [4] and [10], which periodically check the kernel.

3. Implementation

3.1 Requirements

We implemented KRGuard in Linux 2.6.32 as the Linux Kernel Module (LKM) with an Intel Core i5-3470 3.2-GHz CPU. To implement KRGuard, we needed to satisfy the following technical requirements:

- Hooking the transition to the system call handler and collecting a system call number
  - We need to hook the transition to the system call handler to move a control-flow to the function of enabling the LBR. In addition, we need to collect a system call number to judge whether an invoked system call is a monitored system call.
- Hooking a transition to the system call service routine
  - We need to hook the transition to the system call service routine to move a control-flow to the function of checking branch records.
- Collecting branch records using the LBR
  - KRGuard detects kernel rootkits by checking branch records that are recorded by the LBR. Therefore, we need to collect branch records using the LBR.
- Collecting process information
  - If a process is traced, KRGuard outputs the executed program-name and process ID to the user. Therefore, we need to collect the flag that indicates whether tracing occurs, the executed program-name, and process ID.

3.2 Hooking the Transition to the System Call Handler and Collecting a System Call Number

Hooking the transition to the system call handler is implemented by overwriting the SYSENTER_EIP_MSR register. In the SYSENTER_EIP_MSR register, the address of the system call handler is stored. We can move a control-flow to our hook function by overwriting the address of a system call handler stored in the SYSENTER_EIP_MSR register with the address of our hook function. Additionally, KRGuard reads the system call number in the EAX register.

3.3 Hooking a Transition to the System Call Service Routine

Hooking a transition to the system call service routine is implemented by overwriting the address of system call service routines stored in the system call table with the address of a hook function. The system call service routines that correspond to a specific system call are hooked and monitored. (described in Sect. 2.3).

3.4 Collecting Branch Records Using the LBR

The LBR recording is enabled by setting the 0th bit of the MSR_DEBUGCTLA_MSR register (LBR flag), and is disabled by resetting the flag. Branch records are recorded for up to 16 entries, and each entry is indicated by a location number of 0 to 15. The location number indicating the latest branch record is stored in the lower 4 bits of the MSR_LASTBRANCH_TOS register, and KRGuard reads these bits to obtain the location number of the latest branch record. In addition, we can get branch records by reading the LBR stack register. Clearing branch records is implemented by overwriting the LBR stack register with all zeroes.

3.5 Collecting Process Information

If a process is traced, KRGuard outputs the execution program-name and process ID to the user. To achieve this, we collected the following information:

- The flag indicating whether a process is being traced
- Execution program-name
- Process ID

We are able to collect the above information from the

process control block, which is a data structure that contains the information needed to manage a particular process. In Linux 2.6.32, the process control block consists of the thread_info_structure and the task_struct structure. KRGuard evaluates the "flag" variable in the thread_info structure, indicating whether a program is being traced. In addition, KRGuard collects the "comm" and "pid" variables, which store the program-name and process ID, respectively.

4. Evaluation

4.1 Purpose and Environment

The evaluation items and purposes of evaluation are indicated below:

- Detection experiment of kernel rootkit
  - We evaluated the ability of KRGuard to detect kernel rootkits by infecting the target OS with an existing kernel rootkit.
- Performance overhead
  - We measured the overhead per system call incurred by KRGuard. In addition, we measured the processing time of compiling the Linux kernel to evaluate its effect on the performance of real applications.

The evaluation environment is described in Table 1.

4.2 Detection Experiment of Kernel Rootkit

In this evaluation, we used the KBeast [11] program as a real kernel rootkit to infect the target Linux OS. By reviewing our logs, we confirmed that KRGuard was able to detect the presence of the KBeast program. In addition, Fig. 3 depicts the branch records recorded before/after infection with KBeast and shows that the LBR recorded two branch records before infection, and more than or equal to 16 branch records after infection. Since the LBR recorded more than two pieces of branch records, it showed that KRGuard can detect the kernel rootkit.

Table 1 Evaluation environment.

OS kernel	Linux kernel 2.6.32-5 (32bit)
CPU	Intel Core i5-3470 3.2-GHz
Memory	4.0 GB

```
1 :FROM 0xc10030f4  TO 0xf7cab4a3
2 :FROM 0xf7cab0aa  TO 0xc1003078
3 :FROM 0x00000000  TO 0x00000000
4 :FROM 0x00000000  TO 0x00000000
5 :FROM 0x00000000  TO 0x00000000
6 :FROM 0x00000000  TO 0x00000000
7 :FROM 0x00000000  TO 0x00000000
8 :FROM 0x00000000  TO 0x00000000
9 :FROM 0x00000000  TO 0x00000000
10:FROM 0x00000000  TO 0x00000000
11:FROM 0x00000000  TO 0x00000000
12:FROM 0x00000000  TO 0x00000000
13:FROM 0x00000000  TO 0x00000000
14:FROM 0x00000000  TO 0x00000000
15:FROM 0x00000000  TO 0x00000000
16:FROM 0x00000000  TO 0x00000000
```

(a) Before infected with the KBeast

```
1 :FROM 0xf7cb504d  TO 0xf7cab4a3
2 :FROM 0xf7cb501d  TO 0xf7cb5038
3 :FROM 0xc113c76c  TO 0xf7cb501b
4 :FROM 0xc113c764  TO 0xc113c754
5 :FROM 0xc113c764  TO 0xc113c754
6 :FROM 0xc113c764  TO 0xc113c754
7 :FROM 0xc113c764  TO 0xc113c754
8 :FROM 0xc113c764  TO 0xc113c754
9 :FROM 0xc113c764  TO 0xc113c754
10:FROM 0xc113c764  TO 0xc113c754
11:FROM 0xc113c764  TO 0xc113c754
12:FROM 0xc113c764  TO 0xc113c754
13:FROM 0xc113c764  TO 0xc113c754
14:FROM 0xc113c764  TO 0xc113c754
15:FROM 0xc113c764  TO 0xc113c754
16:FROM 0xc113c764  TO 0xc113c754
```

(b) After infected with the KBeast

Fig. 3 Branch records before/after infection with KBeast.

4.3 Performance Overhead

We evaluated the performance overhead per system call incurred by KRGuard by measuring the processing time per system call. The system calls measured were open(), getdents64() and read(). We measured the processing time per open() and getdents64() by taking the average time over 1000 invocations of each call. We measured the processing time per read() by taking the average time over 1000 read attempts of 1KB and the average time over 1000 read attempts of 100KB into the buffer.

Table 2 lists the measurement results for open() and getdents64(), and Table 3 lists the measurement results for read(). According to the results in Tables 2 and 3, the overheads per system call incurred by KRGuard are 0.77 μs–0.80 μs, which are larger than the overheads incurred by Ikegami's method [3] (0.01 μs–0.37 μs in the following environment: Pentium4 3.60-GHz CPU and 4-GB memory). However, compared with other kernel rootkit detection methods, the performance overhead per system call incurred by KRGuard is sufficiently smaller. The overhead per system call is larger than that of Ikegami's method because KRGuard has additional overhead that is generated by reading and writing to the LBR stack register.

Table 4 shows the compiling time of the Linux kernel before/after introducing KRGuard. It shows that the performance overhead for compiling the Linux kernel is 16.6 s (0.74%). Based on this result, we think that the overhead incurred by KRGuard and its effect on a real application's performance are small.

5. Related Work

There are some research studies on control flow integrity (CFI). CFIMon [12] detects violations of the control flow integrity of applications based on hardware support for performance monitoring in modern processors. CFIMon uses a Branch Trace Store (BTS) to obtain all of the branch information of a running application. BTS involves some over-

Table 2 Processing time of open() and getdents64() before/after introducing KRGuard (μs).

System call	Before	After	Overhead
open()	0.39	1.18	0.79
getdents64()	0.07	0.85	0.78

Table 3 Processing time of read() before/after introducing KRGuard (μs).

System call	File size	Before	After	Overhead
read()	1KB	0.24	1.01	0.77
	100KB	4.26	5.06	0.80

Table 4 Processing time of compiling Linux kernel (s).

Before	After	Overhead
2146.43	2162.49	16.06 (0.74%)

head to obtain all branch information.

On the other hand, our target is the kernel. This requires the overhead to be small because all applications are affected by the overhead of the kernel. Thus, the challenge in detecting kernel rootkits to minimize the overhead of the security mechanism and its false positives. Therefore, we leveraged LBR and limited kernel rootkits to be detected. The number of stored branch recodes of LRB is limited; thus KRGuard does not involve large overhead. In addition, the duration of LBR enabled is restricted in the system call handler, and it prevents false positives.

Ge et al. [13] proposed a mostly automated approach to produce and enforce fine-grained CFI policies comprehensively for kernel software with low overhead. This approach can be applied to a wider range of kernel software and prevents more attacks than KRGuard. On the other hand, as mentioned above, because KRGuard deploys the hardware function LBR and limits the target function, the overhead of KRGuard is smaller than that of [13]. In addition, we think that KRGuard can easily be applied to the Linux kernel because the mechanism is very simple.

There are some Return-oriented programming (ROP) exploit run-time prevention techniques using LBR [7], [14], [15]. PathArmor [14] is a context-sensitive CFI approach, and it can restrict the number of illegal control flows for applications. ROPecker [15] and kBouncer are practical ROP prevention techniques that detect abnormal control transfers of ROP execution. These approaches can extensively detect violations of the control flow and control transfer in applications. Although the range of detectable illegal control transfers in KRGuard is small, KRGuard can detect kernel rootkits that modify the control flow of the system call handler with low overhead at run-time.

6. Conclusions

This paper proposed KRGuard, which is a new method to detect kernel rootkits by checking branch records in LBR. By using the LBR, KRGuard can monitor all indirect branches between the invoking system call and the transition to each system call service routine, including indirect branches that do not push data into the kernel stack. In addition, since the LBR is a feature of the CPU and not the OS, KRGuard has high portability to different systems and versions. KRGuard checks branch records every time a system call, monitored by KRGuard, is invoked. Therefore, after an injection with kernel rootkits, KRGuard can detect kernel rootkits immediately. In addition, KRGuard does not prohibit additional kernel modules.

Our evaluation demonstrates that KRGuard can detect the KBeast program, which is a real kernel rootkit. In the evaluation of performance, it is shown that overheads per system call incurred by KRGuard are about 0.77 μs–0.80 μs. In addition, the overhead of compiling the Linux kernel is 16.6 s (0.74%), and we think that the overhead incurred by KRGuard is small.

References

[1] Y. Akao and T. Yamauchi, "Proposal of kernel rootkits detection method by monitoring branches using hardware features," Proc. 2015 IIAI 4th International Congress on Advanced Applied Informatics, pp.721–722, 2015.

[2] Y. Akao and T. Yamauchi, "KRGuard: Kernel rootkits detection method by monitoring branches using hardware features," Proc. 2016 International Conference on Information Science and Security (ICISS), pp.22–26, 2016.

[3] Y. Ikegami and T. Yamauchi, "Proposal of kernel rootkits detection method by comparing kernel stack," IPSJ Journal, vol.55, no.9, pp.2047–2060, 2014 (in Japanese).

[4] X. Wang and R. Karri, "Reusing hardware performance counters to detect and identify kernel control-flow modifying rootkits," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol.35, no.3, pp.485–498, 2016.

[5] N.L. Petroni, Jr, and M. Hicks, "Automated detection of persistent kernel control-flow attacks," Proc. 14th ACM Conference on Computer and Communications Security (CCS '07), pp.103–115, 2007.

[6] Intel, Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B.

[7] V. Pappas, M. Polychronakis, and A.D. Keromytis, "Transparent ROP exploit mitigation using indirect branch tracing," Proc. 22nd USENIX Security Symposium, pp.447–462, 2013.

[8] R. Riley, X. Jiang, and D. Xu, "Multi-aspect profiling of kernel rootkit behavior," Proc. 4th ACM European Conference on Computer Systems (EuroSys '09), pp.47–60, 2009.

[9] Z. Wang, X. Jiang, W. Cui, and P. Ning, "Countering kernel rootkits with lightweight hook protection," Proc. 16th ACM Conference on Computer and Communications Security (CCS '09), pp.545–554, 2009.

[10] X. Wang and R. Karri, "NumChecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters," Proc. 50th Annual Design Automation Conference, no.79, pp.1–7, 2013.

[11] KBeast, available from http://packetstormsecurity.com/files/108286/KBeast-Kernel-Beast-Linux-Rootkit-2012.html (accessed 2016-07-05).

[12] Y. Xia, Y. Liu, H. Chen, and B. Zang, "CFIMon: Detecting violation of control flow integrity using performance counters," Proc. 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '12), pp.1–12, 2012.

[13] X. Ge, N. Talele, M. Payer, and T. Jaeger, "Fine-grained control-flow integrity for kernel software," 2016 IEEE European Symposium on Security and Privacy (EuroS&P), pp.179–194, 2016.

[14] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical context-sensitive CFI," Proc. 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15), pp.927–940, 2015.

[15] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R.H. Den, "ROPecker: A generic and practical approach for defending against ROP attacks," 21st Annual Network and Distributed System Security Symposium (NDSS), 2014.