© 2016 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

# KRGuard: Kernel Rootkits Detection Method by Monitoring Branches Using Hardware Features

Yohei Akao, Toshihiro Yamauchi
*Graduate School of Natural Science and Technology,*
*Okayama University, Okayama, Japan*
*Email: yamauchi@cs.okayama-u.ac.jp*

***Abstract*—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 discuss KRGuard (Kernel Rootkits Guard), 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 hardware features of commodity processors. Our evaluation shows that KRGuard can detect kernel rootkits with small overhead.**

***Keywords*-Security, operating system, kernel rootkit, last branch record**

## I. Introduction

Rootkits are malicious 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. Kernel rootkits modify the operating system (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. Ikegami et al. [1] mentioned that the existing kernel rootkits detection methods can not resolve all of the following problems simultaneously: (1) cannot detect kernel rootkits immediately, (2) cannot keep the expansibility of the OS kernel, and (3) cannot extend to different OS and OS versions. To resolve those problems, Ikegami et al. [1] proposed a method to detect kernel rootkits by checking the kernel stack. 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).

We proposed KRGuard (Kernel Rootkits Guard), which is the new method to detect kernel rootkits resolving all problems (1)-(4) simultaneously [2]. 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.

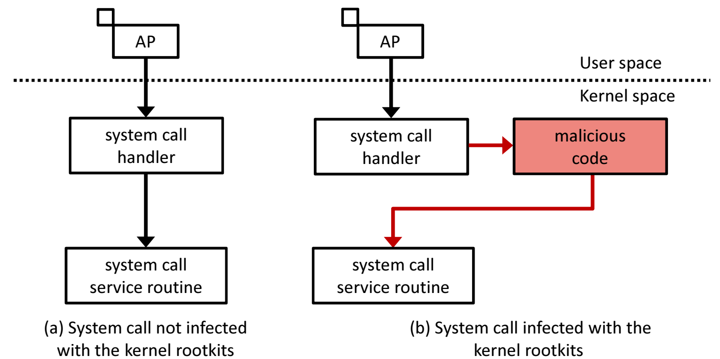


Figure 1. Changes in the control-flow when system call control-flow is modified

In [2], we proposed KRGuard, but we did not describe the implementation and evaluation. In this paper, we describe the implementation of KRGuard as a kernel module on Linux and the evaluation results of KRGuard.

The contributions made in this paper are as follows:

- The implementation of KRGuard on Linux, that can effectively detect kernel rootkits.
- The evaluation of KRGuard. Our evaluation shows that KRGuard can detect the existing kernel rootkit with small overhead.

## II. Design of KRGuard

### A. Concept of KRGuard

KRGuard utilizes the fact that many kernel rootkits make branches that differ from the usual branch path. Previous research [3] indicates that 96% of all kernel rootkits employ control-flow modifications, making branches different from usual. For example, Figure 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 modification. KRGuard uses the Last Branch Record, a recent feature of Intel processors for monitoring branch records in kernel space.

### *B. Last Branch Record*

Last Branch Record (LBR)[4] 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. 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 [5].

### *C. Overview of KRGuard*

KRGuard detects kernel rootkits that modify 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 a 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 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 is abnormal (see Case (2) described in II-D), KRGuard alerts the user.

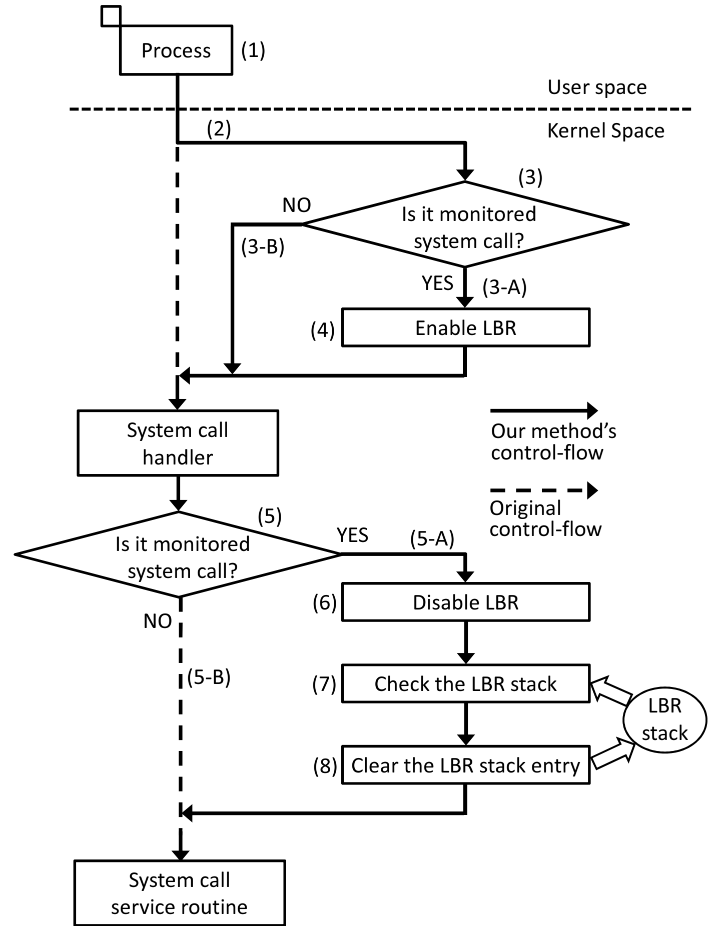


Figure 2. Processing flow of KRGuard

(8) Branch records in the LBR stack is 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 decided by referring [1], [6], and [7].

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

### *D. 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 is 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 equal to two, KRGuard determines that the computer system is not infected with kernel rootkits. When the quantity is greater than two, KRGuard verifies whether or not the process is traced intentionally by the user by outputting process information to the user. If the user intentionally trace 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 the case in which an interruption occurs is an issue that we will consider in the future.

## III. Implementation

### A. Requirements of implementation

We implemented KRGuard to Linux 2.6.32 as the Linux Kernel Module (LKM) with Intel Core i5-3470 3.2GHz 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 indicating whether being traced, executed program-name, and process ID.

### B. 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 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.

### C. 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 a system call service routines stored in the system call table with the address of hook function. The system call service routines that correspond to a specific system call are hooked and monitored. (described in II-C).

### D. 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 up to 16 entries, and each entry is indicated by the 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 and clearing branch records is implemented by overwriting the LBR stack register with all zeroes.

### E. Collecting process information

If a process is traced, KRGuard output 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. On 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.

## IV. Evaluation

### A. Purpose and environment

The evaluation items and the purposes of the 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.

Table I
EVALUATION ENVIRONMENT

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

Figure 3. Branch records before/after infected with the KBeast

- 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 to the performance of real applications.

The evaluation environment is described in Table I.

### B. Detection experiment of kernel rootkit

In this evaluation, we used the KBeast[8] 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, Figure 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 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.

### C. 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 II shows the measurement results of open() and getdents64(), and Table III shows the measurement results of read(). According to the results in Table II and Table III, the overheads per system calls incurred by KRGuard are $0.77\mu$s-$0.80\mu$s, which are larger than the overheads incurred by the Ikegami's method [1] ($0.01\mu$s- $0.37\mu$s in the following environment: Pentium4 3.60GHz CPU and 4GB memory). However, compared with other kernel rootkit detection methods, the performance overhead per system call incurred by KRGuard is sufficiently smaller. Why the overhead per system call is larger than Ikegami's method is that KRGuard has additional overhead that is generated by reading and writing to the LBR stack register.

Table II
PROCESSING TIME OF OPEN() AND GETDENTS64() BEFORE/AFTER INTRODUCING KRGUARD ($\mu$S)

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

Table III
PROCESSING TIME OF READ() BEFORE/AFTER INTRODUCING KRGUARD ($\mu$S)

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

Table IV
PROCESSING TIME OF COMPILING THE LINUX KERNEL (S)

| Before | After | Overhead |
|---|---|---|
| 2146.43 | 2162.49 | 16.06 (0.74%) |

Table IV shows the compiling time of Linux kernel before/after introducing KRGuard. It shows that the performance overhead of compiling the Linux kernel is 16.6s (0.74%). According to this result, we think that the overhead incurred by KRGuard to real application's performance is small.

## V. CONCLUSIONS

KRGuard detects 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 the 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 the 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 $0.77\mu$s-$0.80\mu$s.

In addition, the overhead of compiling the Linux kernel is 16.6s (0.74%), and we think that the overhead incurred by KRGuard is small.

Challenges for future efforts include: handling the case in which an interruption occurs and increasing the variety of kernel rootkits that KRGuard can detect.

## REFERENCES


[1] 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).

[2] 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.

[3] 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.

[4] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B," available from¡http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.html¿ (accessed 2016-07-06).

[5] 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.

[6] 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.

[7] 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.

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