International Journal of Information Security
https://doi.org/10.1007/s10207-020-00514-7

REGULAR CONTRIBUTION

Additional kernel observer: privilege escalation attack prevention mechanism focusing on system call privilege changes

Toshihiro Yamauchi[1] · Yohei Akao[1,2] · Ryota Yoshitani[1] · Yuichi Nakamura[3] · Masaki Hashimoto[4]

Abstract

Cyberattacks, especially attacks that exploit operating system vulnerabilities, have been increasing in recent years. In particular, if administrator privileges are acquired by an attacker through a privilege escalation attack, the attacker can operate the entire system and cause serious damage. In this paper, we propose an additional kernel observer (AKO) that prevents privilege escalation attacks that exploit operating system vulnerabilities. We focus on the fact that a process privilege can be changed only by specific system calls. AKO monitors privilege information changes during system call processing. If AKO detects a privilege change after system call processing, whereby the invoked system call does not originally change the process privilege, AKO regards the change as a privilege escalation attack and applies countermeasures against it. AKO can therefore prevent privilege escalation attacks. Introducing the proposed method in advance can prevent this type of attack by changing any process privilege that was not originally changed in a system call, regardless of the vulnerability type. In this paper, we describe the design and implementation of AKO for Linux x86 64-bit. Moreover, we show that AKO can be expanded to prevent the falsification of various data in the kernel space. Then, we present an expansion example that prevents the invalidation of Security-Enhanced Linux. Finally, our evaluation results show that AKO is effective against privilege escalation attacks, while maintaining low overhead.

Keywords Privilege escalation attack prevention · Operating system · Linux kernel vulnerabilities · Non-control-data attack · System security

1 Introduction

Cyberattacks have been increasing in recent years. One tactic of successful attacks is for attackers to get administrative privilege in the target systems. Although there are various techniques to get the privilege, privilege escalation attacks are a commonly used attack method. A privilege escalation attack exploits operating system (OS) and AP vulnerabilities. Exploiting OS (kernel) vulnerabilities is an especially serious threat to the prevention of privilege escalation attacks, and attacks on OS vulnerabilities have increased [1,2]. The OS kernel consists of a massive amount of code. For example, the number of lines of code in Linux kernel 4.5.3 exceeds 16 million [3]. Therefore, it is difficult to eliminate all vulnerabilities from the OS kernel, and many OS vulnerabilities have been reported thus far [2,4,5].

In a privilege escalation attack, an attacker can promote the privilege of a process to a higher level by exploiting an OS vulnerability. If such a privilege escalation attack succeeds, the attacker can operate the system with a privilege that is higher than the originally assigned one. For example, if an administrator's privilege is acquired by an attacker through a privilege escalation attack, the attacker can avoid access control and gain read/write capabilities for all system information. Thus, this attack type poses a significant security threat to the entire system and countermeasures are required to prevent them.

Existing countermeasures include control-flow integrity (CFI) [6–8]. CFI techniques can prevent attacks that tamper with the control flow of a program. However, they are unable

✉ Toshihiro Yamauchi
yamauchi@cs.okayama-u.ac.jp

1 Graduate School of Natural Science and Technology, Okayama University, 3-1-1 Tsushima-naka, Kita-ku, Okayama 700-8530, Japan

2 NTT Communications Corporation, Tokyo, Japan

3 Hitachi, Ltd., Tokyo, Japan

4 Graduate School of Information Security, Institute of Information Security, Kanagawa, Japan

Published online: 25 June 2020

to prevent non-control-data attacks [9] that alter the data of a program not directly related to its control flow. In addition, Trusted Boot [10] and Integrity Measurement Architecture (IMA) [11] can verify that a program has not been altered at the program startup. However, these two countermeasures cannot prevent attacks that exploit vulnerabilities at run-time. OSs with enhanced security mechanisms, such as Security-Enhanced Linux (SELinux) [12,13], TOMOYO Linux [14], AppArmor [15,16], and Security-Enhanced Android (SEAndroid) [17,18], can restrict damage within the range of the respective policies by dividing the administrative privilege, even if it is acquired by an attacker. However, damage may occur within the ranges of the policies. In addition, it is difficult to not only configure policies for introducing OSs with enhanced security mechanisms, but also to operate them.

In this paper, we describe the characteristics of privilege management in Linux according to our investigation. Privilege management has the following three features.

1. The privilege of the process is stored in the kernel space.
2. To modify data in the kernel space, it is necessary to proceed through a system call.
3. The function of each system call is subdivided. System calls to change privilege information are limited.

Next, this paper proposes the additional kernel observer (AKO), which prevents privilege escalation attacks that exploit OS vulnerabilities. We designed the AKO based on the three features of privilege management. AKO monitors changes in process privileges in the kernel space before and after an invoked system call service routine is processed. It can detect an abnormal privilege change when a process privilege is changed by a system call that does not originally change the privilege. Then, AKO judges that a privilege escalation attack has occurred and prevents it by invalidating the changes to the privilege. AKO can also terminate or suspend the running process to prevent the attack. Thus, AKO can prevent privilege escalation attacks by changing process privileges that are not originally changed in a system call, regardless of the type of vulnerability.

Furthermore, AKO can be expanded to prevent the falsification of various data in the kernel space. We describe the design and implementation of the expansion of observation data for SELinux as an example of the prevention of kernel data falsification.

In this paper, we describe the design and implementation of AKO for Linux x86 64-bit. Evaluation results showed that AKO detected and prevented privilege escalation attacks while maintaining low overhead.

In short, the main contributions of this paper are as follows:

1. We propose a method that can prevent various privilege escalation attacks that exploit OS vulnerabilities through the system call interface. The proposed method can prevent privilege escalation attacks if the attacks are targeted by it, even if they exploit a zero-day vulnerability.
2. The proposed method can be adopted to protect other data in the kernel space. As an expansion of the proposed method, a method that prevents attacks against SELinux is proposed.
3. In addition to its effectiveness against privilege escalation attacks, the proposed method is very simple and the overhead is thus very low. Furthermore, the number of lines of code required for the introduction of AKO is small; hence, modifications do not affect existing functions.

2 Privilege escalation attacks exploiting OS vulnerabilities

2.1 OS vulnerabilities

Many OS kernels are implemented using C and an assembly language, which may contain bugs related to memory management. Consequently, most reported OS kernel vulnerabilities are due to a lack of proper memory management [8,19].

If a vulnerability is found in the OS kernel, a patch must be applied to the kernel to resolve the vulnerability. However, to apply such a patch to the OS kernel, it is necessary to restart the OS in many cases. Thus, it is difficult to apply a kernel patch to a system that needs to operate continually. In addition, to apply patches, it is necessary to download the appropriate patches. Accordingly, applying them to embedded devices is difficult.

For the aforementioned reasons, and based on the premise that OSs have unknown vulnerabilities, it is desirable to deploy in advance a mechanism to prevent attacks that exploit OS vulnerabilities.

2.2 Privilege escalation attacks

A privilege escalation attack enables an attacker to gain illegally elevated access to resources by exploiting a bug, design flaw, etc. A user or an application that obtains a higher privilege than the authorized one can perform originally unauthorized actions. In real attacks, many attackers intend to acquire the authority of the administrator. By acquiring administrative privileges, the attacker can operate the entire system, which can result in information leakages and denial of service.

Privilege escalation attacks are also regarded as threats to mobile devices, such as smartphones and tablets. On the Android, rooting is often performed to gain administrative

privileges for altering critical settings of the target device. The Android uses the Linux kernel, and the rooting is mainly performed by exploiting the Linux kernel vulnerabilities. Once a device is rooted, intellectual property, such as application programs and libraries independently developed by device manufacturers, may be leaked.

2.3 Attacks targeted by AKO

We assume that an attacker can tamper with the data in the kernel space by exploiting a memory corruption vulnerability. In order to emphasize the importance of preventing attacks on critical OS data such as privilege information among memory corruption vulnerabilities, we investigated memory corruption vulnerabilities that could allow privilege escalation reported on the JVN iPedia Vulnerability Countermeasure Information Database [20] in 2019. Approximately 89% of the vulnerabilities reported to be capable of privilege escalation (189 out of 211) are classified as "Critical" or "High" in the CVSSv3 score. This result shows that attacks using vulnerabilities that allow privilege escalation have a serious impact on security. Therefore, it is important to take countermeasures against vulnerabilities that may lead to privilege escalation and focus on data that can detect and prevent privilege escalation. The attacks targeted by AKO are those that exploit Linux kernel vulnerabilities and tamper with the important data in the kernel space such as the privilege data of processes during system call processing.

In addition, since the performance of the OS affects all running applications, the use of an OS security mechanism with a practical overhead is necessary. One of the features of AKO is that it can focus on attacks with high severity and detect them at a practical speed. On the other hand, monitoring all kernel memory has a large overhead. Other attacks against other kernel data due to memory corruption are outside the scope of this research. In order to counter these attacks, it is necessary to utilize other technologies and research results that make the attacks difficult and study new methods.

Because the process privileges are stored in the kernel space and cannot be referenced by user applications in the user space, they cannot be directly tampered with by applications executed at the user level. However, attacks that issue system calls in a sophisticated manner and exploit vulnerabilities in the kernel space can tamper with process privileges stored in the kernel space. Therefore, we focus on the changes to privilege data during system call processing to prevent privilege escalation attacks.

As an example of this attack, we present a privilege escalation attack that exploits CVE-2014-0038. CVE-2014-0038 is a memory corruption vulnerability due to improper parameter checking in the recvmmsg system call. When CONFIG_X86_X32 is enabled, the compat_sys_recvmmsg function in net/compat.c in Linux kernel versions preceding 3.13.2 allows local users to gain privileges via a recvmmsg system call with a specially customized timeout pointer parameter [21]. In a privilege escalation attack that exploits CVE-2014-0038, a recvmmsg system call with sophisticated specially customized parameters is called multiple times. Then, the pointer of the kernel function called in the open system call is changed to the pointer of a kernel function for changing permissions. Subsequently, the exploit code issues the open system call to the target kernel function by referencing the changed pointer. It rewrites the privilege of its process in the open system call, thereby achieving privilege escalation. As a kernel function for rewriting the privilege of its own process, the prepare_kernel_cred() function or commit_creds() is used.

It has been reported that addr_limit, which is a kernel variable indicating the boundary address between the user space and kernel space, is falsified by calling a futex system call with sophisticated specially customized parameters, leading to successful privilege escalation attacks that tamper with the privilege of a process [22]. If the privilege data of a process stored in the kernel space are altered by a privilege escalation attack, as described above, the process can operate the system with a higher authority than the originally conferred authority level.

In the threat model employed in this research, a non-control-data attack was implemented against the important data in the kernel space. The targeted kernel data in the kernel space were privilege data as well as data that were important for security functions. By making non-control-data attacks difficult, it is expected attackers will be hindered from achieving their goals. On the other hand, we assumed attacks involving the rewriting of arbitrary data in the kernel space to be out of scope. However, not all exploitation of OS vulnerabilities can rewrite arbitrary data in the kernel space. Thus, we suppose that an AKO is effective against non-control-data attacks that cannot be prevented by CFI techniques.

To prevent non-control-data attacks, the AKO should be protected, which was done by assuming the CFI technology in the kernel space and kernel address space layout randomization (KASLR) to be deployed in combination with the AKO. The details of AKO protection are described in Sect. 6.4.

3 AKO design

3.1 Concept

The objective of the proposed method is to prevent privilege escalation attacks that exploit OS vulnerabilities. In addition, we aim to prevent privilege escalation attacks due to unknown OS vulnerabilities. First, we investigate privilege

management in Linux, and we clarify its three features as mentioned in Sect. 1.

Based on these features, we assume that the privilege of a process is changed only when a system call with the role of changing a privilege is invoked. However, this will not apply when privilege escalation attacks that exploit Linux kernel vulnerabilities are conducted. For example, in the privilege escalation attack described in Sect. 2.3, the privilege of a process is changed during the processing of the open system call. However, the open system call is not a system call that would change the process privilege. In other words, in a privilege escalation attack that exploits a Linux kernel vulnerability, the process privilege is changed during the processing of a system call that does not originally change the privilege of the process. We assume that this privilege change during system call processing in a privilege escalation attack occurs in many OSs in addition to Linux.

In this paper, we propose AKO to prevent privilege escalation attacks by focusing on privilege changes during system call processing. AKO detects privilege escalation attacks by checking the change of the privilege after system call processing is completed, whereby the invoked system call does not change the privilege of processes. At the time of detection, AKO issues a warning to the administrator. Subsequently, the process can be terminated or the change of the privilege data can be invalidated by overwriting the original privilege data. Consequently, AKO can prevent the damage caused by the attack.

Because the kernel overhead affects all applications, heavy processing involving kernel overhead cannot be implemented to prevent attacks. Therefore, the present objective is to design a method that makes the success of privilege escalation attacks significantly difficult with small overhead and minimal side effects.

Next, we describe the AKO design by considering Linux 4.4 (x86 64-bit) as an example. Note that AKO does not depend on a specific version of the Linux kernel; it can be applied to other versions.

3.2 Privileges for observation

This section describes the process privileges monitored by AKO before and after system call processing. Table 1 lists the privileges (privilege information) monitored by AKO. All privilege information in Table 1 is stored in the kernel space.

The uid group (uid, euid, fsuid, and suid) and the gid group (gid, egid, fsgid, and sgid) store the user identifier and the group identifier, respectively. Such privilege information is used for checking access rights to files and directories, and for verifying whether privileged operations are allowed.

The capabilities field (cap_inheritable, cap_permitted, cap_effective, and cap_ambient) stores flags that indicate whether a process for performing a specific action is permitted [23]. Such capabilities include "performing various network-related operations" and "allowing execution of a chroot system call."

Table 1 Privileges monitored by AKO

Monitored data	Contents
uid	User ID
euid	Effective user ID
fsuid	File system user ID
suid	Saved user ID
gid	Group ID
egid	Effective group ID
fsgid	File system group ID
sgid	Saved group ID
cap_inheritable	Inheritable capabilities
cap_permitted	Permitted capabilities
cap_effective	Actually used capabilities
cap_ambient	Ambient capability set
addr_limit	Highest legal virtual address of user space

In addr_limit, the boundary address between the user space and kernel space is stored. In the normal state, the uid group, gid group, and capabilities are stored in the kernel space, and they cannot be freely rewritten from the user space. When the value of addr_limit is altered by an attacker, the area where the uid group, gid group, and capabilities are stored can be recognized as the user space, not the kernel space. Consequently, the state can be freely rewritten from the user space. It has been reported that addr_limit is falsified by calling a futex system call with sophisticated specially customized parameters, leading to successful privilege escalation attacks that tamper with the privilege of a process [22]. Therefore, the value of addr_limit is also monitored by AKO.

3.3 AKO process flow

The AKO process flow is shown in Fig. 1 and is described below.

First, a process invokes a system call in user mode, and the processing mode is then changed to kernel mode (1). Next, AKO hooks the transition to a system call service routine and moves to the processing of AKO (2). AKO then stores the current privilege information to a reserved saving area before the processing of a system call (3). Next, the invoked system call service routine is executed (4). AKO then hooks the processing immediately after the execution of the system call service routine, and the processing moves to the processing of AKO (5). To check for any changes in privilege information in the system call processing, AKO compares it with the stored privilege information for before the system call processing of Step 3 (6). If AKO detects a change in the

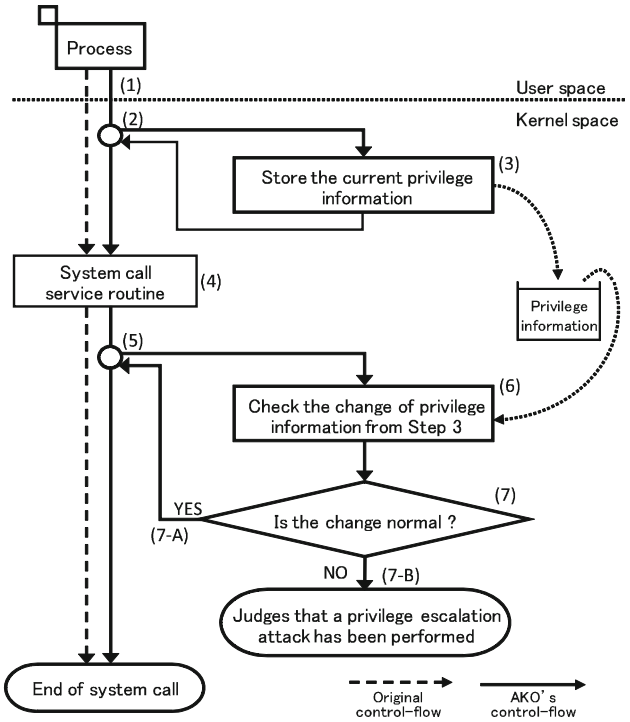

Fig. 1 Process flow of AKO

privilege information, it then checks whether the change is normal (7). If the change in the privilege information is normal, the processing then returns to the original flow and the system call processing terminates because AKO judges that a privilege escalation attack has not been performed (7-a). If the change in the privilege information is not normal, AKO outputs a log indicating that a privilege escalation attack has been detected, and countermeasures against the privilege escalation attack are carried out because AKO judges that a privilege escalation attack has been performed (7-b). Details of the countermeasures are provided in Sect. 3.4.

The abnormal change in privilege information in Step 7 implies that the privilege information, which does not change in each system call process, has been changed. Table 2 lists the system calls that can change the privilege information. From our investigation, among the 313 system calls in Linux 4.4 (x86 64-bit), the 13 system calls listed in Table 2 were found to be capable of changing the privilege information.

AKO determines that an unauthorized change of privilege information occurs when any privilege information of a target process that is not included in Table 2 is changed. For example, because the capset system call is a system call that can change cap_inheritable, cap_permitted, cap_effective, and cap_ambient, when privilege information other than these privileges has been changed before and after the capset system call, AKO judges that the privilege information has been illegally changed.

3.4 Operation after privilege escalation attack detection

We describe countermeasures that are adopted against privilege escalation attacks when AKO detects such attacks. First, AKO uses the Linux auditing system to save an event log when an attack is detected. AKO uses the Linux Auditing System for saving AKO event logs. AKO event logs include the system call number as well as the privilege information before and after the invoked system call. We can track the security events detected by AKO. If the auditing system is not enabled, AKO can save event logs in the kernel log by using the printk() function.

In addition, when a privilege escalation attack is detected, the following countermeasures can be selected and executed to mitigate the effect of the attack.

Invalidation of privilege escalation attacks The purpose of AKO is to protect the target system. In particular, it is important to prevent an attacker from acquiring administrative privileges. Therefore, we believe that invalidating the privilege escalation after AKO detects the attack is very effective in protecting the target system. AKO stores the original privilege information before the system call processing starts. Furthermore, AKO overwrites the privilege information using the stored privilege information when it detects a privilege escalation attack.

Termination of a running process An attacker's process often executes a shell or other malicious program with the root privilege; then, the executed program achieves the attacker's purpose, such as information theft. Terminating a process detected by AKO is effective because doing so can prevent a shell or other malicious program from succeeding in the attack. Invalidating the attack does not terminate the detected process; thus, attackers may continue to attack. On the other hand, process termination can definitely end the attack. However, it is necessary to consider safe termination of the process without harming the system. To prevent kernel processing from being affected, we consider process termination using the SIGKILL signal.

Suspend a running process for analysis In addition to attack prevention, the demand exists for analyzing whether an attack has occurred, or for analyzing the contents of attacks. Thus, AKO can suspend a process that detects a privilege escalation attack in an inexecutable state to analyze the process. Then, the program and memory images can be analyzed to understand the privilege escalation method.

When a privilege escalation attack is detected, AKO invalidates the change performed by the privilege escalation attack. It can then terminate or suspend a running process as an additional countermeasure. These countermeasures must be selected with consideration of the trade-off between security and availability.

Table 2 System calls that may change privileges

System call name	System call number	Privilege can be changed
execve	59	uid, euid, fsuid, suid, gid, egid, fsgid, sgid, cap_inheritable, cap_permitted, cap_effective, cap_ambient, addr_limit
setuid	105	uid, euid, fsuid, suid, cap_inheritable, cap_permitted, cap_effective, cap_ambient
setgid	106	gid, egid, fsgid, sgid
setreuid	113	uid, euid, fsuid, suid, cap_inheritable, cap_permitted, cap_effective, cap_ambient
setregid	114	gid, egid, fsgid, sgid
setresuid	117	uid, euid, fsuid, suid, cap_inheritable, cap_permitted, cap_effective, cap_ambient
setresgid	119	gid, egid, fsgid, sgid
setfsuid	122	fsuid, cap_inheritable, cap_permitted, cap_effective, cap_ambient
setfsgid	123	fsgid
capset	126	cap_inheritable, cap_permitted, cap_effective, cap_ambient
prctl	157	cap_inheritable, cap_permitted, cap_effective, cap_ambient
unshare	272	cap_inheritable, cap_permitted, cap_effective, cap_ambient
setns	308	cap_inheritable, cap_permitted, cap_effective, cap_ambient

4 Expansion of observed data

4.1 Overview

AKO monitors changes in the privilege information shown in Table 2. We assume that the concept of monitoring the change in the data contents in the kernel space before and after a system call processing is effective for kernel space data other than the privilege information.

4.2 Attack prevention against SELinux

As an example of expansion of observed data, we explain the design of attack prevention against SELinux. SELinux enhances system security by introducing a fine-grained access control model called Type Enforcement into the Linux kernel. However, some kernel exploit code disables SELinux by overwriting its data structure [24,25].

To prevent such attacks, AKO monitors SELinux-related data (Table 3). For a running process, sid represents SELinux credential information. In widely used SELinux configurations, some sids can access practically all processes; by overwriting `sid` or `exec_sid`, attackers can obtain these sids. This scenario is almost the same as disabling SELinux. If `selinux_enforcing` is overwritten, SELinux access control is completely disabled. A write or an execve system call may change the data in Table 3. AKO judges that SELinux was attacked if any system calls, excepting write or execve system calls, change the data in Table 3. Table 4 lists system calls that may change the data in Table 3. If system calls not included in Table 4 change the data in Table 3, AKO judges that SELinux was attacked.

Table 3 Data monitored by AKO to prevent attacks against SELinux

Monitored data	Contents
sid	Credential information of SELinux for a running process
exec_sid	sid value is set after execve() system call finishes
selinux_enforcing	identifier indicates SELinux is enable or not

5 AKO implementation

We implemented an extension to Linux kernel 4.4 to prevent privilege escalation attacks as well as attacks against SELinux, as mentioned in Sect. 4. The implementation was carefully performed to disallow bypassing by exploit code. There are two ways to bypass AKO. The first is skipping the check function shown in Step 6 of Fig. 1. The second is overwriting the privilege information saved in Step 3 of Fig. 1. AKO is implemented to prevent such bypass operations throughout the process.

Table 4 System calls that may change the data in Table 3

System call name	System call number	Privilege can be changed
write	1	sid, exec_sid, selinux_enforcing
execve	59	sid, exec_sid, selinux_enforcing

In the system call handler that is executed immediately after a system call is issued, hooks that call AKO are inserted before and after the system call service routine is called. The *before* and *after* hooks are shown in Steps 2 and 5 of Fig. 1. An existing hook framework, such as the Linux Security Modules (LSM) framework [26], is not used; the hooks are embedded directly into the system call handler because the hook framework is easily bypassed by overwriting the function pointer of the hooks.

In the *before* hook, privilege information is saved in the kernel stack, not in the Linux task structure. In terms of the implementation, it is easier to save privilege information in the task structure; however, the address of the task structure is easily obtained and overwritten by accessing the *current* variable of the Linux kernel. Therefore, privilege information is saved in the stack area. Judgment as to whether or not a system call can change the privilege information of a process can be implemented by acquiring the system call number stored in the rax register immediately after the system call has been issued and then comparing it with the contents of Tables 2 and 4.

We implemented the observation method. AKO without SELinux expansion contains 232 lines of code. We implemented AKO in the source code file of the system call handling routine. The range of the source code in which AKO is implemented is very limited, affecting only a small number of lines of code. We suppose that AKO has negligible side effects other than system call processing. This is because AKO is executed only *before* and *after* an invoked system call service routine. Therefore, the modifications do not affect existing functions.

In addition, we implemented the attack prevention against SELinux. The implementation for this expansion checks three data items listed in Table 3. 31 additional lines of code are used for the expansion for SELinux in x86 64-bit, which is a very small addition. Therefore, this result indicates that the amount of code modification required to extend data to be monitored in AKO is small.

6 Evaluation

6.1 Evaluation environment

The evaluation items and evaluation purpose are described below.

Detection of privilege escalation attacks: In the environment in which AKO is applied, we performed multiple types of privilege escalation attacks and evaluated whether AKO could detect them.

Performance measurement: It is anticipated that the implementation of AKO will influence the system performance. Therefore, by comparing the performance before and after AKO implementation, we could benchmark the overhead.

6.2 Prevention experiments for privilege escalation attacks

In the Linux environment (x86 64-bit) used to implement AKO, we performed a privilege escalation attack that exploits a Linux kernel vulnerability, and we evaluated whether AKO could detect the attack. We used five types of exploit codes [21,27–30] for the attack. The potential vulnerabilities varied with the version of the Linux kernel. Each type of exploit code targeted a different vulnerability and thus a different version of the Linux kernel. Then, we conducted an experiment, in which we implemented and tested our detection method for the five versions of the Linux kernel targeted by a different type of each exploit code.

The Common Vulnerabilities and Exposures (CVE) number of the vulnerability exploited by the exploit code used in our detection experiment, as well as the vulnerability outline and results of the detection experiment, are summarized in Table 5. It is evident that AKO could detect privilege escalation attacks exploiting five different vulnerabilities of the Linux kernel.

- In particular, it detected that the uid, gid, and capability groups were changed before and after the sendto system call routine in CVE-2013-1763.
- In CVE-2014-0038, AKO detected that the uid, gid, and capability groups were changed before and after the open system call service routine.
- In CVE-2014-3153, AKO detected that the addr_limit was changed before and after the futex system call service routine.
- In CVE-2016-0728, it detected that the uid, gid, and capability groups were changed before and after the system call service routine of keyctl.
- In CVE-2017-6074, it detected that the uid, euid, fsuid, and suid were changed before and after the system call service routine of recvfrom.

Table 5 Results of the detection experiment of privilege escalation attacks

CVE number	Overview of vulnerability	Kernel version	Detection
CVE-2013-1763	Array index error due to inadequate parameter check in socket()	Linux 3.5.0	✓
CVE-2014-0038	Memory destruction due to inadequate parameter check in recvmmsg()	Linux 3.8.0	✓
CVE-2014-3153	Inadequate address check for re-queuing operation in futex()	Linux 3.10.0	✓
CVE-2016-0728	Use of integer overflow and freed memory in keyctl()	Linux 3.19.0	✓
CVE-2017-6074	Mishandles DCCP_PKT_REQUEST packet data in dccp_rcv_state_process()	Linux 4.4.0	✓

Attacks other than those by the exploit codes used in this detection experiment can also be detected by AKO if they illegally change authority via a system call. As noted above, AKO can prevent privilege escalation attacks before damage occurs to the system by adopting the countermeasures.

6.3 Evaluation of performance overhead

6.3.1 Experimental results of system call overhead

We measured the system call overhead by implementing AKO using the lat_syscall of LMbench 3.0 [31], which is the micro-benchmark suite of the OS. We used a computer with Core i5-3470 3.2 GHz (four cores) and 4.0-GB main memory for the evaluation. The OS's and versions used in the evaluations were Linux 3.10.0 (x86 64-bit).

Table 6 lists the measured results. For the result of open() + close(), the overhead divided by two is written as overhead per the system call. Table 6 shows that the overhead per system call is 0.008–0.036 μs, which is small.

The processing added by the implementation of AKO is limited to a hook of the system call, acquisition of privilege information, and checking whether the privilege information has changed. This is because the fixed overhead occurs per a system call issue. The real overhead in application processing depends on the number of system call issues. For example, for applications that issue 1000 system calls in Linux x86 64-bit, the total overhead is around tens of milliseconds. Processing with a high processing load is not added; hence, we infer that this result is reasonable.

6.3.2 Impact on app performance by introducing AKO

To verify the impact on the application performance, we measured the performance of the web server before and after the implementation of AKO and compared their values. The web server used for the evaluation was Apache 2.4.6. In the evaluation, by using ApacheBench 2.3, we measured the processing time per request when files with sizes of 1 kB, 10 kB, and 100 kB were accessed 10,000 times on a 1 Gbps communication channel. The environment on the server side is the same as the previous evaluations (x86 64-bit). For the client-side environment, the CPU was an Intel Pentium 4 (3.60 GHz) processor with 1 GB RAM; the kernel version was Linux 2.6.32. The number of parallel connections when sending a request was one. The results are listed in Table 7. It is evident that the overhead per request after implementing AKO is approximately 0.002 ms. Moreover, the ratio of the overhead to the processing time before implementing AKO is 0.4% or less.

We also measured and compared the build time of the kernel (Linux 3.10.0) before and after using AKO. The results

Table 6 System call overhead (μs)

System call	Before introduction	After introduction	Overhead
stat()	0.368	0.383	0.015
fstat()	0.099	0.111	0.012
write()	0.105	0.141	0.036
read()	0.078	0.110	0.032
getppid()	0.040	0.048	0.008
open()+close()	1.130	1.190	0.030

Table 7 Processing time per request in Apache web server (ms)

File size (kB)	Before introduction	After introduction	Overhead
1	0.465	0.467	0.002
10	0.638	0.640	0.002
100	1.523	1.525	0.002

Table 8 Processing time in building the kernel (s)

Before introduction	After introduction	Overhead
2669.0	2675.0	6.0 (0.2%)

are listed in Table 8. It is shown that the overhead of the kernel build time after implementing AKO is 6.0 s. In addition, the ratio of the overhead to the processing time before implementing AKO is 0.2%.

The results indicate that AKO does not significantly impact the performance time in a real application.

6.4 Security analysis

AKO stores current privilege information in the kernel stack prior to calling a system call service routine and holds that data in the kernel stack during the system call processing. Thus, to mitigate AKO, an attacker must tamper with both the privilege information of the running process and the stored privilege information. In addition, the attacker must tamper with it during one system call processing. Either altering the privilege information of the process table or altering the stored privilege information would result in the detection of AKO.

To make attacks more difficult, the address for storing privilege information can be randomized. The address can be randomly determined at compile time. In another method, the address can be randomized by using the randomized bits of the kernel address space layout randomization (KASLR). KASLR randomizes the address space layout at each boot. This method extracts the randomized bits in a randomized address, and it randomizes the address for storing privilege information using the extracted bits. After the randomization, attackers must guess the address of the stored privilege information for each targeted kernel. This reduces the possibility of successful attacks against AKO. Thus, we assume that the possibility that AKO would detect the attempts of attacks would increase before the attacks succeed. In addition, KASLR can randomize the address of the text area, which increases the difficulty experienced by an attacker in trying to bypass AKO.

According to our investigation, only a small number of privilege escalation attacks exploiting OS vulnerabilities can execute arbitrary code in kernel mode. In addition, if the arbitrary code can be executed in kernel mode, generating an arbitrary code to mitigate both the randomized address for storing privilege information and KASLR is difficult. Therefore, the number of OS vulnerabilities that can bypass AKO is small.

Combining AKO with CFI techniques is effective for further increasing the difficulty of bypassing AKO. In particular, we suppose that kernel control-flow integrity technique (KCoFI) [8] is effective in preventing the bypass of AKO. Furthermore, methods [32,33] that check the processing flow of system call processing are effective for AKO protection. These methods focus on the flow of system call processing, and can detect abnormal process flow during system call processing. This method detect abnormal process flow by focusing on the flow during system call processing using the last branch record [34]. The method in [32] checks whether the system call service routine was called and executed normally by checking the kernel stack. The method in [33] checks whether the system call service routine was called normally using last branch record [34]. These methods impose low additional overhead because they focus on only system call processing. By expanding these two methods for the calling of AKO functions, it is possible to prevent the bypassing of AKO.

7 Discussion

7.1 Limitations

The proposed method can prevent privilege escalation attacks that tamper with privilege information during system call processing. However, there are privilege escalation attacks that do not tamper with privilege information during system call processing, and these attacks cannot be prevented by AKO. For example, attacks that exploit the vulnerability (CVE-2016-5159), called Dirty COW [35] (reported in October 2016), cannot be prevented by AKO. Dirty COW is a vulnerability in which a race condition occurs during a copy-on-write process, and an unprivileged user can write to a region of read-only memory by exploiting it. Consequently, privilege escalation is achieved, such as by tampering with the contents of /etc/passwd. Attacks that exploit the Dirty COW vulnerability must be addressed by other methods, such as mandatory access control (MAC) systems and Tripwire [36].

7.2 False positives and false negatives

The proposed method prevents privilege escalation attacks by detecting changes in privileges that would not occur under normal conditions while permitting all authority changes that may occur in normal states. Thus, false positives do not occur. However, of the data items (Tables 1 and 3) used by AKO to monitor changes, false positives may occur when monitoring changes in a global variable. For example, because `selinux_enforcing` is a global variable, rather than a variable prepared for each process, the value of `selinux_enforcing` may be changed by other processes in a multi-core environment. However, false positives occurring in the aforementioned case would be very rare, and

Table 9 Types of attacks that can be prevented by CFI and AKO

	Tampering with control flow	Non-control-data attack
CFI	Preventable	Unpreventable
AKO	Only attacks in which privilege information tampered with can be prevented.	Attacks targeting privilege information and other observed kernel data can be prevented

we have not observed a false positive in our environment in such a case thus far.

False negatives occur when an attack that tampers with the authority within the range can occur under normal conditions. For instance, in our method, changes in cap_inheritable, cap_permitted, cap_effective, and cap_ambient are permitted by using the capset system call. Therefore, an attack cannot be prevented if the capset system call is vulnerable; an attacker could exploit the vulnerability and tamper with cap_inheritable, cap_permitted, cap_effective, or cap_ambient using the capset system call. However, because the range of possible attacks is limited, such attacks have become very difficult to execute. For instance, consider an attacker attempting to tamper with uid. In AKO, four system calls, namely execve, setuid, setreuid, and setresuid, are permitted to change uid (see Table 2). Hence, attackers must be able to exploit a vulnerability in one of the four system calls to alter uid. The number of system calls that can be exploited to allow tampering with uid is limited to 1.3% by implementing AKO because there are 313 system calls in Linux kernel 4.4 (x86 64-bit). Hence, it will be more difficult to conduct an attack.

7.3 Application to other OSs

If an OS satisfies the three features mentioned in Section 1, AKO can be applied to the OS. Most monolithic OSs provide their functions through the system call interface and manage privilege information in the kernel space. Therefore, we suppose that AKO can be applied to most monolithic OSs. Even in micro-kernel OSs, if they satisfy the features mentioned in Section 1, AKO can be applied, and privilege escalation attacks can be prevented. We leave application to OSs other than Linux as future work.

8 Related work

8.1 Prevention of privilege escalation attacks

Privilege escalation attacks are performed by an attacker to acquire administrator privilege on a system. Because the goal of these attacks is to gain administrator privilege, countermeasures for these attacks have been studied [37,38]. In [37], the design of privilege separation is proposed. In privilege separation, parts of an application execute with different privileges. This approach is very effective for system services that require privilege.

Privilege escalation attacks can be successful by exploiting OS vulnerabilities. In [38], PrivWatcher is proposed and is a framework for monitoring and protecting the integrity of process credentials and their usage contexts against memory corruption attacks. PrivWatcher also guarantees the "time of check to time of use" (TOCTTOU) consistency. PrivWatcher stores process credentials in a safe region and mediates changes to process credentials. However, PrivWatcher assumes some conditions, such as an isolated domain, kernel protection of MMU operations, and kernel protection against code modification. Thus, a PrivWatcher prototype is implemented in QEMU/KVM. The protection mechanism for process credentialing of PrivWatcher requires a protected memory region and a kernel access control policy. On the other hand, AKO does not need an isolated domain, and we obtain it in the kernel implementation with small overhead and minimal side effects from kernel processing. In addition, AKO can significantly reduce the attack surface for exploiting OS vulnerabilities.

Li et al. [39] verified that the OS kernel paths accessed by popular applications in everyday use contain significantly fewer security bugs than less frequently used paths. Those authors proposed a prototype system that locks an application into using only popular OS kernel paths. The system can prevent the triggering of zero-day kernel bugs. This approach was applied to a virtual machine, and they showed that the prototype was effective in preventing zero-day Linux kernel bugs in a test against an OS-level virtual machine.

8.2 CFI

CFI is a security mechanism that prevents attacks that tamper with a program's control flow [6,7]. By applying CFI, we can prevent the following: execution of code in the stack area, alteration of the return address, and execution of the user space code by the kernel privilege (return-to-user attack [8]). Most CFI techniques target the user space, whereas CFI of the kernel space can be realized through KCoFI [8]. However, recent studies have shown that attacks may be successful against fine-grained CFI. Therefore, applying CFI alone is insufficient for realizing a secure system [40,41]. A non-control-data attack [9] is an attack that CFI can-

not prevent. Such an attack alters data that are indirectly related to the control flow, such as user input, user identifiers, and flags (variables such as `is_admin`). Table 9 lists the types of attacks that CFI and AKO can prevent. Compared to CFI, AKO has the advantage of being able to prevent non-control-data attacks targeting privilege information and other observed kernel data. However, among attacks that alter control flow, AKO can only prevent attacks that change the privilege information. Therefore, system security can be enhanced by combining CFI and AKO.

8.3 Supervisor mode execution protection

Supervisor mode execution protection (SMEP) [42] is a security mechanism provided by Intel CPUs as of the Ivy Bridge architecture. Enabling SMEP prohibits the execution of code stored in the user space memory from entering the supervisor mode. A similar mechanism is available in ARM architecture and is known as Privileged Execute-Never (PXN) [43]. Owing to SMEP and PXN, it is possible to prevent attacks (such as ret2user [1]) that execute the user space code prepared by an attacker from the kernel space. However, attacks that exploit vulnerabilities in the kernel and alter data stored in the kernel space cannot be prevented using these mechanisms. Furthermore, these mechanisms cannot prevent attacks that intentionally execute altered code or data stored in the kernel space. For example, it is impossible to prevent an attack that executes code that manipulates the parameters of a system call and alters the kernel function pointer to point to a different function residing in the kernel space [21].

SMEP and PXN can be considered as types of CFI because they prevent illegal control flow from the kernel space to the user space. We believe that the security of the system can be enhanced by combining AKO with SMEP and PXN.

8.4 MAC system, Linux security module

A MAC system is a security mechanism that realizes MAC and the principle of least privilege. Typical MAC systems include SELinux [12,13], TOMOYO Linux [14], and AppArmor [15,16]. SEAndroid [17,18], which is an extension of SELinux for the Android, has been included since Android 4.3. By applying the MAC system with an appropriate policy, even if administrative privileges are acquired by an attacker, the damage can be limited within the range permitted by the policy. MAC systems are realized using LSM [26]. LSM hooks processing before a system call starts its processing and transfers it to the hook function in the kernel space. MAC systems perform access control within the hook function of LSM.

The major difference between AKO and LSM is that LSM hooks only before system call processing, whereas AKO hooks before and after system call processing. With only hooks before system call processing, the processing contents of the system call cannot be monitored, and access control can only be performed based on the execution state before system call processing. Therefore, after the processing of the LSM hook function, vulnerabilities can be exploited during system call processing, and even if an illegal processing is executed, it cannot be detected. In [44], recent attacks whereby an attacker exploits a vulnerability of the kernel and invalidates LSM have been identified. By contrast, AKO can check for changes in privilege information during system call processing by hooking before and after the system call processing, preventing privilege escalation attacks that exploit vulnerabilities.

While AKO is realized by rewriting the kernel, LSM is advantageous in that the hook function group can be introduced as a kernel module. In the future, we may be able to implement the hook function in AKO as a kernel module.

8.5 Kernel integrity protection

Trusted Boot [10] and IMA [11] check at program startup that the program has not been altered. Thus, the integrity of the kernel at system startup can be protected. However, attacks that exploit vulnerabilities at run-time cannot be prevented with these architectures.

The kernel integrity protection method at run-time [45–47] prevents code injection into the kernel space by checking that illegal code is not inserted into the kernel space during program execution. These techniques can protect data that are not changed during system execution, such as the code area of the kernel space. However, it is not possible to prevent tampering of data that can be changed depending on the execution state, such as privilege information. For example, the privilege information constitutes data that can be changed even in a normal state. Even if the information is changed, we cannot simply judge that it is illegal. To detect an unauthorized change of data that can be changed during system execution, it is necessary to consider the execution state. Therefore, AKO focuses on the fact that the privilege information can be changed only by specific system call processes. Moreover, by considering the type of system call executed, AKO determines changes of privilege information as being authorized or unauthorized. Thus, AKO prevents privilege escalation attacks by detecting unauthorized changes in privilege information during process execution.

9 Conclusion

In this paper, we proposed AKO, a method that prevents privilege escalation attacks from exploiting Linux kernel vulnerabilities. We described its implementation and presented evaluation results.

AKO is based on the following features of privilege data management and system calls. First, the privilege of the process is stored in the kernel space. Second, to modify data in the kernel space, it is necessary to proceed through a system call. Third, the function of each system call is subdivided.

By applying AKO, we can prevent a privilege escalation attack that changes privilege information that should not be changed in the original system call processing, regardless of the content of the vulnerabilities. In addition, by the introduction of AKO to a system in advance, it is possible to prevent privilege escalation attacks without applying a security patch to the kernel for resolving a vulnerability.

Furthermore, by examining data structures in the kernel space and system call processing, the data to be monitored by AKO can be expanded for preventing unwanted kernel data falsification. In this paper, as an extension of AKO, a mechanism for preventing attacks that disable SELinux was proposed.

We implemented AKO and performed experiments involving privilege escalation attacks using exploit codes available on the web. The evaluation results showed that AKO could detect multiple types of privilege escalation attacks. Thus, AKO can adopt countermeasures against privilege escalation attacks before damage to the system occurs.

The performance evaluation results of LMbench showed that the overhead per system call was 0.008–0.036 μs, which is small. The ratio of the overhead to the processing time in the web server before implementing AKO was 0.4% or less. In addition, the ratio of the overhead to the processing time in the kernel build before implementing AKO was 0.2%. From these results, we can conclude that AKO is effective while having a negligible impact on performance.

In the future, we will consider countermeasures against attacks with memory corruption vulnerabilities that were not considered in this research.

Acknowledgements This work was partially supported by JSPS KAKENHI Grant Number JP19H04109.

Compliance with ethical standards

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Kemerlis, V. P., Portokalidis, G. and Keromytis, A.D.: kGuard: lightweight kernel protection against return-to-user attacks. In: Proceedings of USENIX Security '12, pp. 459–474 (2012)
2. Kemerlis, V.P., Polychronakis, M., Keromytis, A.D.: ret2dir: rethinking Kernel isolation. In: Proceedings of USENIX Security '14, pp. 957–972 (2014)
3. Linux Counter. https://www.linuxcounter.net/statistics/kernel
4. Niu, S., Mo, J., Zhang, Z., et al.: Overview of linux vulnerabilities. In: Proceedings of SCICT 2014, pp. 225–228 (2014)
5. Chen, H., Mao, Y., Wang, X., et al.: Linux kernel vulnerabilities: state-of-the-art defenses and open problems. In: Proceedings of APSys '11, No. 5 (2011)
6. Abadi, M., Budiu, M., Erlingsson, U., et al.: Control-flow integrity. In: Proceedings of CCS '05, pp. 340–353 (2005)
7. Petroni, Jr., N.L., Hicks, M.: Automated detection of persistent kernel control-flow attacks. In: Proceedings of CCS '07, pp. 103–115 (2007)
8. Criswell, J., Dautenhahn, N., Adve, V.: KCoFI: complete control-flow integrity for commodity operating system kernels. In: Proceedings of IEEE S&P '14, pp. 292–307 (2014)
9. Chen, S., Xu, J., Sezer, E.C., et al.: Non-control-data attacks are realistic threats. In: Proceedings of USENIX Security '05, pp. 177–192 (2005)
10. Trusted Computing Group: Trusted Boot. http://www.trustedcomputinggroup.org/trusted-boot/
11. Integrity Measurement Architecture (IMA). https://sourceforge.net/projects/linux-ima/
12. Loscocco, P., Smalley, S.: Integrating flexible support for security policies into the linux operating system. In: Proceedings of the FREENIX Track: USENIX ATC 2001, pp. 29–42 (2001)
13. NSA/CSS: Security-Enhanced Linux. https://www.nsa.gov/what-we-do/research/selinux/
14. TOMOYO Linux. https://tomoyo.osdn.jp/index.html
15. Cowan, C., Beattie, S., Kroah-Hartman, G., et al.: SubDomain: parsimonious server security. In: Proceedings of LISA '00, pp. 355–368 (2000)
16. AppArmor security project wiki. http://wiki.apparmor.net/index.php/Main_Page
17. Smalley, S., Craig, R.: Security enhanced (SE) Android: bringing flexible MAC to Android. In: NDSS 2013 (2013)
18. Security Enhancements (SE) for Android. http://seandroid.bitbucket.org/
19. Szekeres, L., Payer, M., Wei, T., et al.: SoK: eternal war in memory. In: Proceedings of IEEE S&P '13, pp. 48–62 (2013)
20. JVN iPedia. https://jvndb.jvn.jp/en/
21. Exploit Database, Linux Kernel 3.4 < 3.13.2 (Ubuntu 13.04/13.10) - 'CONFIG_X86_X32=y' Local Root Exploit (3). https://www.exploit-db.com/exploits/31347/
22. Exploiting the Futex Bug and uncovering Towelroot. http://tinyhack.com/2014/07/07/exploiting-the-futex-bug-and-uncovering-towelroot/
23. Hallyn, S.E., Morgan A.G.: Linux capabilities: making them work. In: Proceedings of Linux Symposium, pp. 163–172 (2008)
24. Exploit Database, Nexus 5 Android 5.0—Privilege Escalation. https://www.exploit-db.com/exploits/35711/

25. grsecurity: super fun 2.6.30+/RHEL5 2.6.18 local kernel exploit. https://grsecurity.net/~spender/exploits/exploit2.txt
26. Wright, C., Cowan, C., Smalley, S., et al.: Linux security modules: general security support for the linux kernel. In: Proceedings of USENIX Security '02, pp. 17–31 (2002)
27. Exploit Database, Linux Kernel 3.7.10 (Ubuntu 12.10 x64) - 'sock_diag_handlers' Local Root Exploit (2). https://www.exploit-db.com/exploits/24746/
28. Exploit Database, Linux Kernel 3.14.5 (RHEL / CentOS 7) - 'libfutex' Local Root Exploit, https://www.exploit-db.com/exploits/35370/
29. Exploit Database, Linux Kernel 4.4.1—REFCOUNT Overflow/Use-After-Free in Keyrings Privilege Escalation (1)C https://www.exploit-db.com/exploits/39277/
30. kernel-exploits/poc.c. https://github.com/xairy/kernel-exploits/blob/master/CVE-2017-6074/poc.c
31. LMbench—Tools for Performance Analysis. http://www.bitmover.com/lmbench/
32. Ikegami, Y., Yamauchi, T.: Proposal of kernel rootkits detection method by comparing kernel stack. IPSJ J. **55**(9), 2047–2060 (2014). (in Japanese)
33. Yamauchi, T., Akao, Y.: Kernel rootkits detection method by monitoring branches using hardware features. IEICE Trans. Inf. Syst. **E100–D**(10), 2377–2381 (2017)
34. Intel, Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B
35. DIRTY COW. https://dirtycow.ninja/
36. Tripwire Inc.: Tripwire. https://www.tripwire.org/
37. Provos, N., Friedl, M., Honeyman, P.: Preventing privilege escalation. In: Proceedings of USENIX Security '03 (2003)
38. Chen, Q., Azab, A. M., Ganesh, G., et al.: PrivWatcher: non-bypassable monitoring and protection of process credentials from memory corruption attacks. In: Proceedings of ASIA CCS '17, pp. 167–178 (2017)
39. Li, Y., Dolan-Gavitt, B., Weber, S., et al.: Lock-in-pop: securing privileged operating system kernels by keeping on the beaten path. In: Proceedings of USENIX ATC '17, pp. 1–13 (2017)
40. Carlini, N., Barresi, A., Payer, M., et al.: Control-flow bending: on the effectiveness of control-flow integrity. In: Proceedings of USENIX Security '15, pp. 161–176 (2015)
41. Evans, J., Long, F., Otgonbaatar, U., et al.: Control Jujutsu: on the weaknesses of fine-grained control flow integrity. In: Proceedings of CCS '15, pp. 901–913 (2015)
42. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1, http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf
43. ARM Cortex-A Series Programmer's Guide for ARMv8-A. http://infocenter.arm.com/help/topic/com.arm.doc.den0024a/DEN0024A_v8_architecture_PG.pdf
44. Looking at the local Linux kernel privilege escalation. http://blog.siphos.be/2013/05/looking-at-the-local-linux-kernel-privilege-escalation/
45. Azab, A. M., Ning, P., Shah, J., et al.: Hypervision across worlds: real-time kernel protection from the ARM TrustZone secure world. In: Proceedings of CCS '14, pp. 90–102 (2014)
46. Riley, R., Jiang, X., Xu, D.: Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing. In: Proceedings of RAID '08, pp. 1–20 (2008)
47. Seshadri, A., Juk, M., Qu, N., et al.: SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In: Proceedings of SOSP '07, pp. 335–350 (2007)