IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received July 1, 2018, accepted August 5, 2018, date of publication August 21, 2018, date of current version September 7, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2866498

PrivGuard: Protecting Sensitive Kernel Data From Privilege Escalation Attacks

WEIZHONG QIANG, JIAWEI YANG, HAI JIN, (Senior Member, IEEE), AND XUANHUA SHI, (Senior Member, IEEE)

Services Computing Technology and System Lab, Cluster and Grid Computing Lab, Big Data Security Engineering Research Center, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China

Corresponding author: Weizhong Qiang (wzqiang@hust.edu.cn)

This work was supported in part by the National Basic Research Program of China (973 Program) under Grant 2014CB340600, in part by the National Natural Science Foundation of China under Grant 61772221, and in part by the Shenzhen Fundamental Research Program under Grant JCYJ20170413114215614.

ABSTRACT Kernels of operating systems are written in low-level unsafe languages, which make them inevitably vulnerable to memory corruption attacks. Most existing kernel defense mechanisms focus on preventing control-data attacks. Recently, attackers have turned the direction to non-control-data attacks by hijacking data flow, so as to bypass current defense mechanisms. Previous work has proved that non-control-data attacks are the critical threat to kernels. One of the important purposes of these attacks is to achieve privilege escalation by overwriting sensitive kernel data. The goal of our research is to develop a lightweight protection mechanism to mitigate non-control-data attacks that compromise sensitive kernel data. We propose an approach that enforces data integrity of sensitive kernel data by preventing the illegal write to these data to mitigate privilege escalation attacks. The main challenge of the proposed approach is to validate the modification of sensitive kernel data at runtime. The validation routine must verify the legitimacy of the duplicated sensitive data and ensure the credibility of the verification. To address this challenge, we modify the system call entry point to monitor the change of the sensitive kernel data without any change to Linux access control mechanism. Then, we use stack canaries to protect the duplication of sensitive kernel data that are used for integrity checking. In addition, we protect the integrity of sensitive kernel data by forbidding illegal updates to them. We have implemented the prototype for Linux kernel on Ubuntu Linux platform. The evaluation results of our prototype demonstrate that it can mitigate privilege escalation attacks and its performance overhead is moderate.

INDEX TERMS Kernel, non-control-data, credential, privilege escalation.

I. INTRODUCTION

Over the past decades, memory corruption vulnerabilities in kernels of modern commodity operating systems have been a hot topic for the researchers all the time. Since the security of OS kernels is one of the preconditions for many security solutions, attackers have paid more attention to kernel exploits. Through those memory corruption vulnerabilities, attackers can execute arbitrary codes in the kernel or even take full control of the whole system. To protect the kernel from memory corruption exploits, researchers have proposed several kernel-specific defense mechanisms, such as *Supervisor Mode Execution Protection* (SMEP), *Kernel Address Space Layout Randomization* (KASLR), *Kernel Control Flow Integrity* (KCFI), etc. SMEP prevents the execution of the code located in a user-mode page while the operating system is running in a higher privilege level [1]. KASLR makes it much more difficult for attackers to find gadgets in the kernel address space [2]. KCFI introduces a solid solution to prevent subverting the kernel's control flow [3], [4].

Since the defenses against code injection or code reuse attacks [5] that base on manipulating control-data have evolved into mature mechanisms step by step, attackers turn their attention to non-control-data attacks [6]. Researchers have proposed several exploit techniques, such as *Data-Oriented Exploits* and *Data-Oriented Programming* [7]. As we known, attackers usually manipulate the non-control data to achieve privilege escalation without subverting any

control flow of the kernel, for example, overwriting the credentials of the current process (`uid`, `gid`, etc.) with the ones of the root user.

To defend against non-control-data attacks, researchers have proposed several solutions from different aspects. These solutions render the non-control data either impossible to be manipulated [8]–[13], or harder to be located in the kernel [14], [15]. However, all these solutions suffer from at least one of the limitations as follows: high performance overhead, the necessity for higher-privileged execution modes (e.g., hypervisors), dependency on the support of specific hardware features, or incompatibility with the kernel.

The goal of this paper is to provide a practical defense solution for the security-critical non-control data in the kernel to prevent privilege escalation attacks. We achieve this by enforcing data integrity of data structures that involves with access control mechanisms, such as structure `cred`, `mm_struct`, `task_struct`, `addr_limit` (in `thread_info`), and so on. We present a framework called PrivGuard to protect the sensitive data in kernel from being compromised. PrivGuard monitors the modification of the sensitive data by hooking the system calls. Then, PrivGuard saves a duplication of the sensitive data before the invoking of a system call, and uses this duplication for integrity check before the returning of this system call. Stack canaries are used to protect the duplicated sensitive data from be overwritten by attackers. As for data integrity, PrivGuard enforces it by preventing the illegal write to the sensitive kernel data.

We implement a prototype for Ubuntu Linux platform and evaluate its practicality and effectiveness. The performance evaluation results show that it has an acceptable impact on operating system, and an average overhead of 9% on system call on x86 system. The impact on I/O and computation latency is negligible. The real-world attack test case shows that PrivGuard can defend against data-only privilege escalation attacks effectively. In summary, we make the following contributions in this paper:

- We propose a practical method to monitor the modification of sensitive data in kernels by hooking the system calls without changing the existing linux access control mechanisms, and leverage stack canary to protect the duplicated sensitive data.
- We present PrivGuard, a framework that enforces data integrity of kernels' sensitive data by preventing the illegal write to the data.
- We implement a prototype of PrivGuard for Linux kernel. The performance evaluation results show that it incurs an average overhead of 9% on system call and nearly has no impact on I/O and computation. The performance overhead for applications is negligible. The security evaluation for the real-world non-control-data attack shows that PrivGuard can defend it effectively.

We organize the remainder of this paper as follows. Section II briefly introduces kernel privilege escalation attacks. Section III defines the problem scope of this paper.

```
1 void __attribute__((regparm(3))) payload() {
2         commit_creds(prepare_kernel_cred(0);
3 }
```

Listing 1. Privilege escalation payload.

Section IV describes the detailed system design. Section V presents the implementation of PrivGuard. Section VI dicusses the limitation of our work and the future works. Section VII illustrates the evaluation results. Section VIII briefly introduces related work. Section IX presents the conclusion.

II. BACKGROUND

In this section, we introduce the background of privilege escalation attacks by manipulating the security-sensitive data in the kernel. Over the past few years, attackers have paid more attention to the kernel. One of the most common attack vectors is to exploit memory corruption in the kernel to manipulate the control-data to hijack the control flow, or to corrupt the non-control data to tamper with the data flow. In what follows, we briefly introduce these kernel attacks that achieve privilege escalation.

A. PRIVILEGE ESCALATION WITH CONTROL-DATA ATTACKS

The kernel control-data attack is a kind of attack that exploits memory corruption to redirect the kernel's control flow to malicious codes injected by the attacker or the fragments of existing codes chained together by the attacker. The former is called code injection attack, which can be prevented by *Supervisor Mode Execution Prevention* (SMEP), or *Privileged Execute Never* (PXN). The latter is known as the code-reuse attack [5]. The ROP attack is a novel and prevail code-reuse attack technique. In a typical ROP attack for privilege escalation, the kernel control flow is diverted to a user-space address which contains the privilege escalation payload, as illustrated in Listing 1. The privilege escalation payload will allocate a new credential with root privilege and apply it to the current process.

To deal with ROP attacks, researchers put forward a novel technique named *Control Flow Integrity* (CFI) [3], [4]. CFI validates each indirect control flow transfer in the kernel whenever it happens to ensure that the kernel's control flow conforms to the valid control flow graph (CFG) which is acquired by static analysis in advance. However, CFI can also be bypassed with control-data attacks. For example, attackers can overwrite each callback function pointer of SELinux to the default one to disable SELinux, which doesn't violates the policy of CFI.

B. PRIVILEGE ESCALATION WITH NON-CONTROL-DATA ATTACKS

An attack that leverages a memory corruption vulnerability to tamper with security-critical data, rather than redirect the control flow, is a non-control-data attack. Such kinds of attacks are firstly introduced by Chen *et al.* [6] to reveal that they

can cause dangerous security problem. Hu *et al.* [7] prove that non-control-data attacks are Turing-complete. Moreover, due to not bending the control flow transfer of the program, non-control-data attacks can bypass current defense mechanisms that are designed mainly for preventing control-data attacks. Therefore, non-control-data attacks are serious threats to Linux kernel. Non-control-data attacks make privilege escalation by directly compromising sensitive kernel data structure. For example, the `cred` data structure of a process could be located in the kernel address space, and then be overwritten with the one of the privileged process, so as to escalate the privilege of the original process.

Several solutions have been proposed to defend against non-control-data attacks [8], [9], [11], [12], [14], [15]. However, existing defense mechanisms for preventing privilege escalation suffer from some limitations as mentioned in Section I.

```
ssize_t write_pipe(void *readbuf, void *
  writebuf, size_t count) {
    int pipefd[2];
    ssize_t len;
    pipe(pipefd);
    write(pipefd[1], writebuf, count);
    len = read(pipefd[0], readbuf, count);
    if (len != count) {
        printf("___FAILED READ @ %p : %d %d\n"
, writebuf, (int)len, errno);
        while (1) {
            sleep(10);
        }
    }
    read(pipefd[0], readbuf, count);
    close(pipefd[0]);
    close(pipefd[1]);
    return len;
}
```

Listing 2. A function that writes the data referred by `writebuf` to the location pointed by `readbuf`.

III. PROBLEM SCOPE

In this section, we specify the problem scope of our work. First, we describe the threat mode and assumptions. Then, we provide some examples that modify the kernel's data to explain the motivation.

A. THREAT MODEL AND ASSUMPTIONS

The aim of our research is to prevent privilege escalation attacks, which exploit memory corruption vulnerabilities to manipulate security-sensitive data in the kernel. We only take the attacks that come from unprivileged programs in user-space into consideration. Therefore, the attacks that comes from malicious drivers and kernel rootkits are out of scope, since they are at the same privilege level as the kernel, and thus hard to defend against.

Specifically, we make the following assumptions:

Memory Corruption: There is at least one memory corruption vulnerability in the kernel that allows attackers to gain the ability of reading and writing arbitrary memory.

User Space: There is no restriction on attackers in the user space. Attackers can execute any code in the user space and invoke kernel API function.

Kernel Attacks Defense: There are already some defense mechanisms against code injection attacks deployed in the kernel, such as SMEP [1]. There are also some defense mechanisms against code-reuse attacks deployed in the kernel, such as *Control Flow Integrity* [3], [4], *Kernel Address Space Layout Randomization* [2], or *Code-Pointer Integrity* [16]. However, all these defense mechanisms cannot prevent non-control-data attacks.

B. MOTIVATION EXAMPLE

Based on the threat model and assumptions above, we leverage a real-world Linux kernel vulnerability, CVE-2014-3153 [17], to manipulate kernel non-control data to make privilege escalation for the current process. CVE-2014-3153 reports a bug in the system call `futex()` which provides the fast user-space locking. There are two vulnerabilities in the kernel functions `futex_requeue`, `futex_lock_pi` and `futex_wait_requeue_pi`. By leveraging these vulnerabilities, attackers can cause futex variables to have waiters but no owner, the so-called dangling pointer. By filling the function stack, attackers can modify the data in the waiter `rt_waiter` on the stack to control the node of the waiting list in the kernel. Then, they can acquire the ability of writing arbitrary kernel address through inserting nodes into the waiting list to change `addr_limit` with `0xffffffff`.

After overwriting `addr_limit`, as shown in Listing 2, attackers can use the function `write_pipe` to overwrite the memory in the kernel space that `readbuf` points to with the prepared data referred by `writebuf`, by invoking the system calls `read` and `write`. In practice, the content pointed by `readbuf` is normally some security-sensitive data, which thus will be manipulated.

To achieve privilege escalation, attackers usually choose to compromise the process credentials which are typical security-sensitive data. The `cred` structure stores the credentials for a process, and the privilege level of the process is determined by it. Therefore, it is the primary objective in the privilege escalation attacks. For example, Listing 3 shows DAC check in Linux kernel. If attackers overwrite the kernel data like `fsuid`, they can bypass the DAC checks and access sensitive files.

In order to manipulate the `cred` structure, we need to get the location of it at first. Since the `rt_waiter` is on the kernel stack, the address of `thread_info` can be acquired by `rt_waiter & 0xffffe000`. Then, through parsing `thread_info`, the address of `task_struct` and `cred` can be located as `thread_info->task_struct` and `task_struct->cred` step by step.

As shown in Figure 1, there are three ways to get local privilege escalation with the function in Listing 2.

I The address of `cred` can be obtained by referring `task_struct->cred`. Then, attackers can directly

```
static int acl_permission_check(struct inode *
    inode, int mask) {
  unsigned int mode = inode->i_mode;
  if (likely(uid_eq(current_fsuid(), inode->
    i_uid)))
    mode >>= 6;
  else {
    ...
    if (in_group_p(inode->i_gid))
      mode >>= 3;
  }
  if ((mask & ~mode & (MAY_READ | MAY_WRITE |
    MAY_EXEC)) == 0)
    return 0;
  return -EACCES;
}
```

Listing 3. Code snippet in Linux kernel for discretionary access control check.

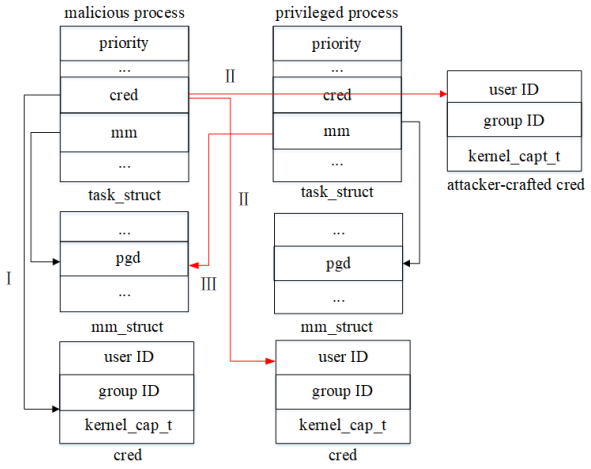

FIGURE 1. Three ways to achieve local privilege escalation.

overwrite the content of `cred`, for examine, `uid=0`, `gid=0`, and so on;

II Attackers can overwrite the pointer of `cred` in the `task_struct` to make it point to a fake or an existing one with root privilege, whicn makes the `cred` describe a malicious process. This changes their context of use.

III The structure `mm_struct` describes the memory map of the process. By parsing `task_struct->mm`, attackers can get the pointer that contains the address of `mm_struct`. Through overwriting the one of attacker-controlled process with the one of privileged process, the context of privileged process is corrupted. The malicious code can run in the privileged context.

IV. SYSTEM DESIGN

In this section, we firstly elaborate the main idea of the proposed defense system. Then, we describe the funtionalities in details. Figure 2 shows a overview of the architecture of our system.

A. MAIN IDEA

As we known, Linux uses two operating modes, kernel mode and user mode. Only trusted core components of operating system run in the kernel mode, and other components and applications run in the user mode. A user-space application must invoke system calls on its behalf to access the kernel data. Thus, the kernel data is not directly accessible to user-space applications, including user-space attack codes.

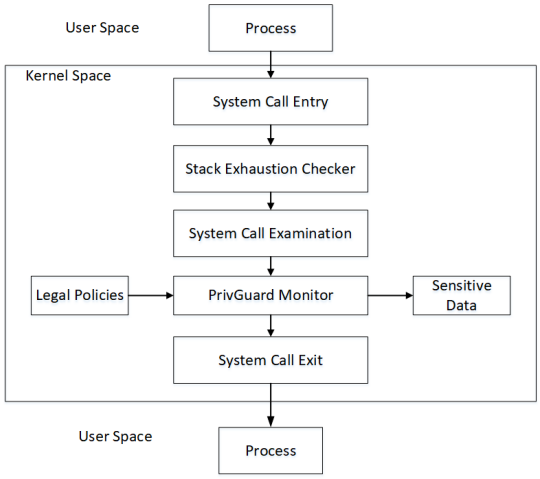

FIGURE 2. The PrivGuard at runtime.

Therefore, we design the system based on system call hooking to monitor the modification of sensitive kernel data. In the implementation of operating systems, those security-sensitive data are only supposed to be modified by some specific system calls, such as `sys_seteuid` and `sys_setcap`, with pre-defined permission-checking rules that will forbid unprivileged users to make privilege escalation through modifying those data. Therefore, the system calls can be divided into two categories: the ones that can change the sensitive kernel data legitimately, and the ones that should have no right to change those sensitive data. We only focus on the latter category to monitor the illegitimate change of the sensitive kernel data and check the integrity of them.

In addition, the system call interfaces of different Linux distributions are seldom changed. Thus, our system needs little change to port from one Linux distribution to another, which means it has good compatibility and portability.

B. SYSTEM CALL HOOKING

Since system calls are the gate that allows a user-mode application to access the sensitive kernel data, we hook them to ensure that the sensitive data is not compromised during their execution.

Listing 4 shows the entry for Linux system calls for x86 platform, which is defined in `entry_32.S` of kernel files. As shown in Listing 4, OS sets up the registers, pushes the syscall number saved in `eax` to kernel stack, and then calls `sys_call_table` to start the invocation of system call. We modify the system call entry point and add two functions to hook the system call, named `syscall_pre()` and `syscall_post()`.

Once a user-space application invokes a system call, the function `syscall_pre()` will be executed at first.

```
ENTRY(system_call)
...
pushl_cfi %eax
SAVE_ALL
GET_THREAD_INFO(%ebp)
testl $_TIF_WORK_SYSCALL_ENTRY,TI_flags(%
  ebp)
jnz syscall_trace_entry
cmpl $(NR_syscalls), %eax
jae syscall_badsys
syscall_call:
...
call syscall_pre
...
call *sys_call_table(,%eax,4)
...
call syscall_post
...
```

Listing 4. Entry point for system calls on x86 platform.

During its execution, this system call will be determined whether to be legitimate to access the sensitive kernel data. Then, a copy of the sensitive kernel data will be duplicated to be used for integrity checking afterwards. Before the return of the system call, the function `syscall_post()` will be invoked, which is in charge of checking the data integrity of the sensitive kernel data which is not supposed to be modified by this system call.

C. SENSITIVE DATA DUPLICATION

PrivGuard prevents illegal write to sensitive kernel data. In order to verify the data, they need to be duplicated and saved on the other place. We duplicate these data when system call starts and verify the modification of them when system call returns.

Each process has a process control block (PCB) to maintain the information about the process in the kernel. In Linux, the PCB is carried by the `task_struct` data structure. The process stores its information in `task_struct`, such as process descriptor, process identifier, process memory management, and so on. Process credentials are stored in another kernel data structure `cred`, which is referred as a pointer in the `task_struct` structure. These data structures are critical to the privilege level of a process. Thus, we systematically examine these data structures and classify the data fields that need to be protected into the following three categories:

I. Identities. The `cred` structure includes user identities and group identities, such as `uid`, `gid`, `euid`, `egid`, etc. If the `uid`/`gid` of a process is zero, it means that this process has root privilege and full access to all files in the system.

II. Capability. The capability of a process represents the operations that the process are allowed to perform. `cap_permitted` indicates the capabilities that the process has. `cap_effective` is the capabilities that the process actually uses at the moment. `cap_inheritable` is the capabilities that child process can inherit. If the capabilities of a process are manipulated by attackers, the ordinary user can perform operations that only the root user has the permission to perform, such as changing the `uid` of the process.

III. Pointers. In `task_struct`, there are several pointers referring to some sensitive data structures, such as the pointer to `cred`, and the pointer to `mm_struct` which refers to the structure used for process memory management. Attackers can manipulate those pointers to refer to a privileged process, so as to achieve privilege escalation.

We combine these sensitive data fields together and put them into a unified structure. When a user-space application invokes system call, the hook function `syscall_pre()` stores the duplication of the security-critical data into the unified structure. Afterwards, `syscall_post()` uses it for checking the integrity of the security-critical data. The duplicated sensitive data should push on the kernel stack in advance for the hook function to use it. To avoid conflicting with the data that already exist in the kernel stack, we choose a safe place that is unused yet. This place is set at a fixed offset to the address of `thread_info`. We obtain the address of `thread_info` by the value of current `esp` minus kernel stack size. Then add a constant value to result above as the stack grows from high address to low address.

The size of the sensitive data duplication is much smaller (about 80 bytes) compared to the size of kernel stack (8 KB). It will not have a bad impact on the execution of system calls or kernel functions. In addition, during the runtime of micro-benchmarks, the operation system does not run into any faults or cash. Moreover, the kernel has a limit to the nest depth of system calls or kernel functions, so the space of kernel stack will not be used up in normal situations.

D. STACK EXHAUSTION CHECKING

Since the duplication of sensitive data was kept on the kernel stack, we should ensure that there has enough space for saving data. What's more, these data shouldn't be overwritten before being duplicated. Therefore, we add a check to prevent stack exhaustion and the overwrite to sensitive data.

In Linux, each process has two different kinds of stacks, task stack in the user mode and kernel stack in the kernel mode. The kernel stack is used by the kernel at the execution of the system calls. As shown in Figure 3, kernel saves the `thread_info` structure at the end of the kernel stack. For x86 architecture, kernel stack size is usually limited to 8 KB. Thus, the kernel stack space can be exhausted if the nesting depth of system calls is too large or a function uses a pretty large stack frame. In such case, the structure `thread_info` can be manipulated, probably with the data crafted by the attacker. As mentioned in Section III, `thread_info` contains some sensitive data, for example, data pointer of `task_struct`. These data are usually objectives for attackers. Previous work has demonstrated such a vulnerability [18] can be exploited by attackers to achieve privilege escalation [19].

To avoid the circumstances mentioned above, we examine the status of the kernel stack at the granularity of system calls.

Before the execution of each system call, we make sure that the current stack pointer is higher than the address of the `thread_info` plus a fixed offset. The fixed offset is used to reserving space for saving the content of registers and the duplication.

E. PROTECTION OF DUPLICATED SENSITIVE DATA

Securing the duplicated sensitive data is necessary, because it is critical for the integrity checking when the system call returns. Since we save these duplicated data in the kernel stack, we secure them by inserting stack canaries close to them and checking the integrity of canaries before using them.

The canary is a data field that is inserted into the stack to protect stack from memory corruption attacks [20]. If the attacker exploits a stack overflow vulnerability to overwrite the data on the kernel stack, the integrity of the canary will be compromised at first.

As shown in Figure 3, a canary is inserted right before the duplicated sensitive data to ensure the integrity of the data. Each time before using the duplicated sensitive data, the integrity of canary will be checked at first to validate the data.

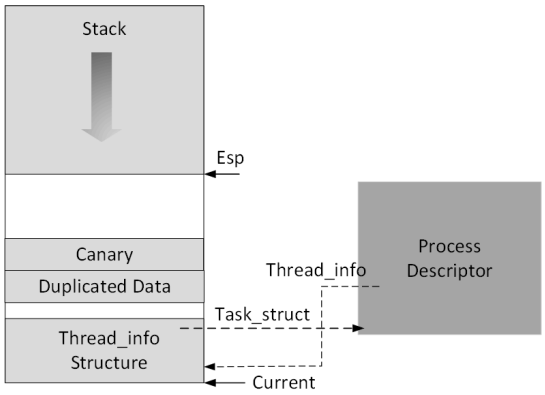

FIGURE 3. Kernel stack layout.

F. DATA INTEGRITY VERIFICATION

The four sections above have established the base of preventing privilege escalation attacks: monitoring the modification of the sensitive kernel data and ensuring the integrity of the duplicated sensitive data. Therefore, we can enforce the data integrity policy over these data to prevent privilege escalation attacks at runtime. PrivGuard can adapt any kind of policy to protect them. For exmaple, Android operating system reserves a set of `uid/gid` for root and system privilege processes (0-10000 normally). PrivGuard can prevent the `uid/gid` of a process changing from the unprivileged set to the privileged set.

In this paper, we adapt a general-purpose data integrity policy to prevent privilege escalation attacks. We prohibit the value of the sensitive kernel data that represents the privilege from being changed from non-privileged to privileged, such as `uid` being changed from non-zero to `0`. But there are some exceptions: a process that executes setuid-to-root binaries, such as `passwd`, will experience the privilege change; a process with the capability of `CAP_SETUID` can change its user identifiers.

1) PREVENTING PRIVILEGE ESCALATION ATTACKS

PrivGuard enforces data integrity policy over the sensitive kernel data to prevent privilege escalation attacks. First of all, we need to give a definition about the operations that cause policy violation.

As mentioned in Section IV-C, `cred` structure contains user identity and group identity data fields: `uid`, `euid`, `suid`, `fsuid`, `gid`, `egid`, `sgid`, and `fsgid`. If an unprivileged process modifies its user identity and group identity to root user identity, we consider this as an illegal privilege escalation, which needs to be terminated.

As for capabilities, taking `cap_effective` for example, once an unprivileged process illegitimately sets some bits of its own `cap_effective` to `1`, it can perform some privileged operations constrained by those bits from that time on. We also consider this as some illegal privilege escalation.

Moreover, the `cred` and `mm_struct` structures are referred as pointers in `task_struct` structure. We prevent the targets of these pointers from being changed from unprivileged to privileged.

2) LEGITIMATE PRIVILEGE ESCALATION

With above runtime data integrity policy adopted, we can prevent attackers manipulating the sensitive kernel data to achieve privilege escalation for the current process. However, there are two exceptions that we should take into consideration:

I Privileged process. In Linux operating system, a process with the capabilities of `CAP_SETUID` can change its user identifiers with the ones of root privilege through system call `sys_setresuid`. For these situations, we define a boolean value to indicate this change is legitimate, and we don't check the data integrity of the sensitive data.

II Setuid-to-root binaries. Another exception that privilege escalation probably happen is the execution of the setuid-to-root binaries. When a setuid-to-root binary is executed, the process is executed as the owner of the binary. Therefore, the privilege level of the process can change from non-root to root. However, when the process executes setuid-to-root binary, it doesn't invoke the system call to change its user identifiers. Thus, PrivGuard will not be triggered to check this change. Since our work is to prevent attacks that achieve privilege escalation by manipulating the sensitive kernel data, attacking setuid-to-root binary is out-of-scope. In addition, previous work has already provided solutions to defend against attacks that aim at setuid-to-root binary [21], [22].

G. ENSURING CONTEXT CONSISTENCY

To this point, PrivGuard can protect sensitive data from the first type of attack mentioned in Section III which directly manipulates the sensitive data. In order to defend against the last two types of attacks described in Section III which corrupts the context of the process by manipulating the pointers, PrivGuard must ensure that these sensitive data describe the same process when using and verifying them. In other words, we must enforce consistency of the context of these data.

There are two contexts in Linux operating system that represent the current process. The first one is the process descriptor, `task_struct` structure. Every process has its own process descriptor. The second one is the page global directory (`pgd`) of the address space of the process, which is unique for each process and saved on the register `cr3`. PrivGuard ensure the consistency of these two contexts. We defines a new structure (`priv_context`) to record the address of `task_struct` and the `pgd`, which is referenced as a pointer in the `cred` structure. To protect `priv_context` from being corrupted by attackers, we save it in read-only region. Since user-space processes access the sensitive data by system calls, the context consistency checks will be performed at the execution of each system call. In PrivGuard, these checks happen in the hooking functions of system calls. In details, PrivGuard checks whether the content of the new structure `priv_context` correctly conforms to the current `task_struct` and `pgd` saved on `cr3`.

V. IMPLEMENTATION

In this section, we present the implementation of PrivGuard on Ubuntu Linux x86 platform.

A. MODIFICATION OF SYSTEM CALL ENTRY POINT

The Linux x86 kernel has two system call entry points: `int 0x80` for interrupt-based system calls and `syscall` for fast system calls. As shown in Listing 5, we modify the system call entry point to invoke the function `syscall_pre` before system call entering and the function `syscall_post` before system call returning.

To call a C function from assembly code, we should use the macro `asmlinkage` as a modifier at the definition of the C function, which means that the arguments for this C function are passed by stack in x86. The registers are used to save some important data during the execution of system calls. For example, the system call number is loaded into register `eax`. And `eax` is also used for saving the return value of the system call. Therefore, to avoid the content of registers being trashed by the invocation of the hook function, we push the content of registers on the kernel stack before invoking the function, and pop the data from the stack to the corresponding registers after the execution of the function. We also push the function arguments on the kernel stack to pass it to the hook function.

B. CANARY INSERTION AND CHECK

Canaries are stored in the higher address next to the sensitive data, so attacks that exploit buffer overflow vulnerability need to overwrite the canaries at first. We generate the canaries with the XOR result of a random value and the sensitive data saved on the stack. As long as the canaries or the sensitive data were overwritten by attackers, it can be detected by checking the data integrity of the inserted canaries.

```
1   ENTRY(system_call)
2 ...
3   pushl_cfi %eax
4   SAVE_ALL
5   GET_THREAD_INFO(%ebp)
6   testl $_TIF_WORK_SYSCALL_ENTRY,TI_flags(%
      ebp)
7   jnz syscall_trace_entry
8   cmpl $(NR_syscalls), %eax
9   jae syscall_badsys
10 syscall_call:
11 ...
12  pushl_cfi %eax
13  movl $-THREAD_SIZE, %eax
14  addl %esp, %eax
15  addl $0x098, %eax
16  pushl_cfi %eax
17 ...
18  call syscall_pre
19 ...
20  popl_cfi %eax
21  popl_cfi %eax
22 ...
23  call *sys_call_table(,%eax,4)
24 ...
```

Listing 5. Modifications for one system call entry point.

The random value is acquired by adopting the Time Stamp Counter (TSC) as the random value generator. The low 32-bit of the TSC value changes frequently (TSC on 1GHz CPU overflows and is reset per 4.3s), thus we can treat it as a random value. The TSC value can be obtained by the unprivileged instruction RDTSC. We perform bitwise XOR of the random value and the sensitive data and push both the TSC value and XOR result on the stack. We verify the canaries through performing bitwise XOR of the sensitive data and the XOR result above. The result should be same as the TSC value saved on the stack.

Integrity checking of canaries is inserted before each conditional branch of data integrity checking that uses the sensitive data to make decision. If canaries have been overwritten before conditional branch performs, the result of data integrity checking could be wrong. Therefore, we consider the process as a malicious program and send a signal to notify the operating system to terminate it.

C. CONTEXT CONSISTENCY CHECK

The `priv_context` is saved in the read-only region, we modify the linker script `vmlinux.lds.S` of x86 architecture to add a section for saving it. When using the `priv_context`, we verify that it belongs to this section. To check the context consistency, each process/thread needs to have its own `cred`. Since `cred` structure can be shared in the same Linux thread group, we disable the sharing and ensure that each `task_struct` links to a unique `cred`.

Although the system calls are designed to be used by user-space process, some kernel threads may also invoke the

system calls. The kernel has a unique address space. The operating system uses the same kernel address space of the previous process when scheduling a kernel thread, thus the `pgd` saved on `cr3` doesn't change. Likewise, in the interrupt, the register `cr3` will record the `pgd` of the interrupted process, thus the `pgd` saved on `cr3` also doesn't change. According to the above facts, PrivGuard checks whether the operating system is scheduling a kernel thread or in the interrupt to avoid false positives.

VI. DISCUSSION

We discuss the potential attacks which can bypass the proposed defense mechanism, PrivGuard. We also introduce the possible future work.

A. USE-AFTER-FREE ATTACKS

For temporal memory corruption vulnerabilities, such as use-after-free (UAF), our design may fail to deal with them. For instance, if the structure `cred` of a process is improperly freed and reallocated to another process with root privilege later, the former process will gain root privilege. However, our design can still mitigate the exploitation of UAF vulnerabilities to some extent. For example, PingPong Root [23] exploits a UAF vulnerability to control a dangling function pointer. Attackers leverage it to overwrite `addr_limit`, and then corrupt the process credentials. However, attackers need to invoke system call `close` to trigger this exploit, which can be detected by PrivGuard.

B. DMA ATTACKS

DMA (direct memory access) can directly access arbitrary physical address, so attackers can manipulate sensitive data in memory by using it. We haven't implemented any defense mechanism against it yet. However, previous work [24] has leveraged IOMMU (input/output memory management unit) to defend against DMA attacks.

C. CODE INJECTION ATTACKS

PrivGuard inserts checks before and after the execution of system calls by modifying the kernel code. Therefore, if the attacker tampers with the inserted code, PrivGuard will probably be bypassed. However, researchers have proposed several mechanisms to protect kernel code integrity, such as code-pointer integrity, control flow integrity, etc. We have assumed the kernel have adopted at least one of the mechanisms to ensure kernel code integrity in Section III-A. Therefore, attackers cannot compromise the inserted code.

D. PORTABILITY ON OTHER PLATFORM

Although we implement our prototype on x86 architecture, the techniques used by PrivGuard are generic to most other architecture platform. Therefore, porting PrivGuard builds to other commodity platform is possible, such as x86-64, AArch32, and AArch64. Some architecture-dependent codes need extra work, such as the modification of system call entry. However, different architectures have similar system call entry, so the extra work is little. As for stack canary, there are some registers like TSC registers, Performance Monitors Control Registers on ARMv8. The rest techniques are both architecture-independent.

VII. EVALUATION

In this section, we illustrate evaluation results of the proposed defense prototype PrivGuard. We evaluate the performance overhead incurred by PrivGuard. We perform all these experiments on a desktop computer with Intel Core i7-4720 2.6GHz processor and 4GB memory, which runs with Ubuntu Release 14.04 x86 with Linux kernel version 3.13.0.

A. MICRO-BENCHMARKS

1) UnixBench

The first experiment we conduct is to evaluate PrivGuard using `Byte UnixBench` with version 5.1.3 [25]. As shown in Table 1, the second column is the overhead percentage of the kernel with PrivGuard compared to a unmodified kernel. What's more, we also include evaluation results from PrivWatcher, which provides protection for process credentials, for comparison. And the third column is the relative overhead percentage of PrivWatcher [9]. The results show that the performance overhead of some test cases is negligible with the value less than 1%, while that of `execl`, `pipe throughput`, `pipe-based context switching`, `process creation` and `system call` is not negligible with the value between 3.1% and 17.3%.

TABLE 1. The results of `UnixBench`.

Test Case	PrivGuard(%)	PrivWatcher(%)
Dhrystone	1.0	-0.1
Whetstone	1.0	-1.1
Execl	6.1	23.3
File Copy 1024	0.2	0.0
File Copy 256	0.4	0.0
File Copy 4096	-0.3	-0.9
Pipe Throughput	16.0	1.3
Pipe-based Context Switching	15.1	/
Process Creation	3.1	22.3
Shell Scripts (8)	1.0	11.0
System Call	17.3	-0.5

Test cases of `execl` and `process creation` both involve with changing the process credentials by invoking system calls, which will trigger the invocation of the system call hook functions. As mentioned in Section IV-A, some system calls can change the sensitive data legitimately, and thus the data integrity checks caused by those system calls do not need to be performed. This situation is applicable to the two cases above, which explains their performance overhead is lower than the results of test cases `pipe throughput`, `pipe-based context switching` and `system call` which need to verify the data integrity because the system calls invoked belong to the second category. As we see from the table, PrivGuard incurs lower overhead than PrivWatcher in test case of `execl`, `process`

TABLE 2. The results of LMBench.

	Native(ms)	PrivGuard(ms)	Overhead	KENALI [11] Overhead	SALADS [15] Overhead	KCoFI [3] Overhead
null syscall	0.0533	0.0686	1.29x	1.00x	1.15x	2.50x
read	0.1121	0.1246	1.11x	/	1.28x	/
write	0.1239	0.1389	1.12x	/	1.29x	/
stat	0.4641	0.4708	1.01x	/	/	/
open/close	1.0328	1.0332	1.00x	2.76x	1.23x	2.47x
select	3.0118	3.1978	1.06x	1.42x	1.18x	1.56x
singal install	0.1743	0.1972	1.13x	1.30x	/	2.14x
singal catch	0.9046	0.9277	1.02x	2.23x	/	0.92x
fork+exit	159.68	181.50	1.14x	2.18x	1.17x	3.53x
fork+execv	627.64	728.06	1.16x	2.26x	/	3.15x
page fault	1.7168	1.7436	1.01x	1.71x	1.09x	1.11x
mmap	6291	6539	1.03x	1.63x	1.11x	3.29x

creation and shell scripts. In test case of pipe throughput, pipe-based context switching and system call, PrivGuard has higher overhead than PrivWatcher.

2) LMBench

We evaluate PrivGuard's impact on system calls in details by using LMBench version 3. Table 2 illustrates the results of LMBench [26]. The system call latency is the average of the results of 10 experiments. The second and third columns show the execution time of the system calls on the unmodified kernel and the kernel with PrivGuard, respectively. For comparison, we also include evaluation results from KENALI [11], which enforces data flow integrity for kernel; SALADS [15], which provides kernel data structure layout randomization; and KCoFI [3], which enforces complete CFI over the whole Linux kernel.

As we can see from Table 2, the performance overhead incurred by PrivGuard in average is about 9%, which is moderate. Especially, the performance overhead for stat, open/close, singal catch, page fault and mmap is pretty low, which can be neglected. Compared to other solutions, the performance overhead is much lower than them. Therefore, our solution that enforces data integrity to prevent privilege escalation attacks is practical.

B. APPLICATION BENCHMARKS

The third experiment is to evaluate the performance overhead of the applications. We conduct the experiment with file compression applications bzip2 and gzip, web server benchmark tool ApacheBench, and kernel compile benchmark script kcbench. We use bzip2 and gzip to compress a 212MB file and measure the time that the compression takes. We set ApacheBench to run 100 POST requests (with 100KB) to the server to measure the processing time per request. We also use kcbench to compile Linux kernel with version 3.13.0 and measure the time it takes.

Table 3 shows the results of application benchmarks. As the results show, PrivGuard introduces little overhead to user applications. Among all the overhead, the highest overhead is 0.99% while the lowest overhead is 0.4%, which is all below 1%. Thus, we believe that the overhead of PrivGuard is negligible on the applications.

TABLE 3. The results of application benchmarks.

Program	Native	PrivGuard	overhead	increase
bzip2	32.600s	32.854s	0.254s	0.7%
gzip	9.600s	9.695s	0.095s	0.99%
ApacheBench	2.2292ms	2.2379ms	0.008ms	0.4%
kcbench	515.09s	518.30s	3.21s	0.6%

C. SECURITY EVALUATION ON PRIVILEGE ESCALATION ATTACKS

We leverage two real-world examples to demonstrate the capability of PrivGuard to defend against privilege escalation attacks. These examples are based on CVE-2014-3153 [17] and CVE-2015-1805 [27].

CVE-2014-3153 reports a flaw in the Linux kernel's futex subsystem through version 3.14.5. The futex_requeue function does not ensure that calls have two different futex addresses, which allows attackers to make privilege escalation via a crafted FUTEX_REQUEUE command that facilitates unsafe waiter modification. The experimental result shows that this attack is prevented at the execution of system call sys_read when the attack tries to change the cred structure of the process illegally.

CVE-2015-1805 reports a bug in the Linux kernel implementation of vectored pipe read and write functionality before version 3.16. The kernel doesn't take the some corner cases that data isn't synchronized during copying into consideration, which leads to out-of-bounds copying of I/O vector array. Therefore, this memory corruption vulnerability allows attackers to escalate their privileges on the system via a crafted application. Our experimental result shows that this attack is prevented at execution of system call sys_read when trying to overwrite the capabilities of the current process.

Therefore, PrivGuard is effective to prevent privilege escalation attacks.

VIII. RELATED WORK

In this section, we elaborate a variety of defense mechanisms for non-control-data attacks.

A. DATA INTEGRITY

PrivWatcher moves process credentials to the safe region and makes the region non-writable to Linux kernel [9]. Then, all the changes to process credentials are mediated by PrivWatcher. It prevents privilege escalation attacks by enforcing data integrity on process credentials. However, PrivWatcher needs to adopt a hypervisor to obtain secure isolated execution domain, which introduces the new attack surfaces to the whole system. For example, the trusted computing base of Xen contains more than ten-million lines of code. By the year of 2015, there are 135 CVEs in Xen, 89 CVES in KVM, and 234 CVEs in VMware. PrivWatcher also doesn't protect the `kernel_cap_t` structure.

Sentry ensures the data integrity of security-critical kernel data by enforcing data access restrictions [12]. It partitions kernel data structures to move all non-sensitive fields to a new structure, which is then referred as a pointer in the original structure. Sentry allocates the original structure that contains sensitive fields into the protected memory pages. Sentry verifies write accesses by checking whether the instruction pointer belongs to the trusted region (kernel address space). This check is inadequate against kernel-level memory corruption, and Sentry is hypervisor-based.

B. DATA FLOW INTEGRITY

DFI is a generic solution to defend against both control-data attacks and non-control-data attacks [8]. DFI generates the data-flow graph via static analysis before the execution of programs. Then, it instruments the program to check whether the flow of data at runtime conforms to data-flow graph. Recently, Song *et al.* [11] applies DFI in Linux kernel. It uses point-to analysis to discover the sensitive data, and then leverages hardware-specific features to achieve data flow isolation between the sensitive data and the non-sensitive data. Considering the performance overhead, Kenali only enforces fine-grained DFI over the sensitive data.

C. DATA SPACE RANDOMIZATION

Since attackers still need to acquire the address of some non-control data pointers, data space randomization can mitigate non-control-data attacks [2], [14], [15]. However, a fine-grained data space randomization may lead to a high performance overhead because all the data structure should be randomized frequently.

D. DYNAMIC TAINT ANALYSIS

DTA is a technique to keep track of tainted data from untrusted sources when the program executes, and then to detect attacks if tainted data is used in a sensitive way [10]. As a use case of DTA in network, it will tag all data coming from the network as tainted, track their propagation, and alert the user when those data are used in a way that could compromise system integrity. Because of the intensive tracking caused by DTA, it will incur high performance overhead.

E. MEMORY SAFETY

Memory safety prevents memory corruption attacks at runtime. Cyclone [28] and CCured [29] store bounds information for each pointer, which is used for bounds checking to enforce memory safety. SoftBound improves such method by storing the bounds metadata into a disjointed space [30]. However, a complete memory safety enforcement at runtime will incur high performance overhead, for example, 116% average overhead incurred by SoftBound.

IX. CONCLUSION

In this paper, we design and implement PrivGuard to protect the sensitive kernel data from privilege escalation attacks. We propose a technique to hook system calls to monitor the modification of the sensitive kernel data, verify the data integrity of those data at runtime, and leverage stack canary to protect the duplicated sensitive data. We implement the PrivGuard prototype on Ubuntu 14.04 with kernel version 3.13.0, and evaluate the practicality and effectiveness of it. The experiment results indicate that its performance overhead is acceptable, and it can defend against privilege escalation attacks effectively.

REFERENCES

[1] S. Fischer. (2011). *Supervisor Mode Execution Protection*. [Online]. Available: https://www.ncsi.com/nsatc11/presentations/wednesday/emerging_technologies/fischer.pdf

[2] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *Proc. 21st USENIX Secur. Symp. (USENIX Secur.)*, 2012, pp. 475–490.

[3] J. Criswell, N. Dautenhahn, and V. Adve, "KCoFI: Complete control-flow integrity for commodity operating system kernels," in *Proc. IEEE Symp. Secur. Privacy*, May 2014, pp. 292–307.

[4] X. Ge, N. Talele, M. Payer, and T. Jaeger, "Fine-grained control-flow integrity for kernel software," in *Proc. IEEE Eur. Symp. Secur. Privacy*, Mar. 2016, pp. 179–194.

[5] N. Carlini and D. Wagner, "ROP is still dangerous: Breaking modern defenses," in *Proc. 23rd USENIX Secur. Symp. (USENIX Secur.)*, 2014, pp. 385–399.

[6] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *Proc. 14th USENIX Secur. Symp. (USENIX Secur.)*, 2005, pp. 177–192.

[7] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *Proc. IEEE Symp. Secur. Privacy*, May 2016, pp. 969–986.

[8] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Proc. 7th Symp. Oper. Syst. Design Implement.*, 2006, pp. 147–160.

[9] Q. Chen, A. M. Azab, G. Ganesh, and P. Ning, "PrivWatcher: Non-bypassable monitoring and protection of process credentials from memory corruption attacks," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, 2017, pp. 167–178.

[10] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2005, pp. 1–43.

[11] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee, "Enforcing kernel security invariants with data flow integrity," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2016.

[12] A. Srivastava and J. Giffin, "Efficient protection of kernel data structures via object partitioning," in *Proc. 28th Annu. Comput. Secur. Appl. Conf.*, 2012, pp. 429–438.

[13] Y. Ding, T. Wei, H. Xue, Y. Zhang, C. Zhang, and X. Han, "Accurate and efficient exploit capture and classification," *Sci. China Inf. Sci.*, vol. 60, no. 5, p. 052110, 2017.

[14] S. Bhatkar and R. Sekar, "Data space randomization," in *Proc. 5th Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*. Berlin, Germany: Springer, 2008, pp. 1–22.
[15] P. Chen, J. Xu, Z. Lin, D. Xu, B. Mao, and P. Liu, "A practical approach for adaptive data structure layout randomization," in *Proc. Eur. Symp. Res. Comput. Secur.* Cham, Switzerland: Springer, 2015, pp. 69–89.
[16] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *Proc. 11th USENIX Symp. Oper. Syst. Design Implement.*, 2014, pp. 147–163.
[17] *CVE-2014–3153*. Accessed: Feb. 15, 2018. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-3153
[18] *CVE-2010–3848*. Accessed: Feb. 15, 2018. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-3848
[19] *Econet Privilege Escalation*. Accessed: Feb. 15, 2018. [Online]. Available: https://www.exploit-db.com/exploits/17787/
[20] C. Cowan *et al.*, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proc. 7th USENIX Secur. Symp. (USENIX Secur.)*, 1998, pp. 63–78.
[21] B. Jain, C.-C. Tsai, J. John, and D. E. Porter, "Practical techniques to obviate setuid-to-root binaries," in *Proc. 9th Eur. Conf. Comput. Syst.*, 2014, pp. 8:1–8:14.
[22] L. Litty, H. A. Lagar-Cavilla, and D. Lie, "Hypervisor support for identifying covertly executing binaries," in *Proc. 17th USENIX Secur. Symp. (USENIX Secur.)*, 2008, pp. 243–258.
[23] W. Xu and Y. Fu, "Own your Android! Yet another universal root," in *Proc. 9th USENIX Conf. Offensive Technol.*, 2015, pp. 1–6.
[24] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," in *Proc. 21st ACM SIGOPS Symp. Oper. Syst. Princ.*, 2007, pp. 335–350.
[25] *UnixBench*. Accessed: Feb. 15, 2018. [Online]. Available: https://github.com/kdlucas/byte-unixbench
[26] L. W. McVoy and C. Staelin, "lmbench: Portable tools for performance analysis," in *Proc. USENIX Annu. Tech. Conf.*, 1996, pp. 279–294.
[27] *CVE-2015–1805*. Accessed: Feb. 15, 2018. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1805
[28] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of C," in *Proc. USENIX Annu. Tech. Conf.*, 2002, pp. 275–288.
[29] G. C. Necula, S. McPeak, and W. Weimer, "CCured: Type-safe retrofitting of legacy code," in *Proc. 29th ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, 2002, pp. 128–139.
[30] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "SoftBound: Highly compatible and complete spatial memory safety for C," in *Proc. Conf. Program. Lang. Design Implement.*, 2009, pp. 245–258.

WEIZHONG QIANG received the Ph.D. degree in computer engineering from the Huazhong University of Science and Technology (HUST), China, in 2005. He is currently an Associate Professor with HUST. He has authored or co-authored about 30 scientific papers. His topics of research interests include system security about virtualization and cloud computing.

JIAWEI YANG received the B.Sc. degree in computer science from the University of Electronic Science and Technology, China, in 2015, and the M.Sc. degree in information security from the Huazhong University of Science and Technology, China, in 2018, respectively.

HAI JIN (SM'06) received the Ph.D. degree in computer engineering from the Huazhong University of Science and Technology, China, in 1994. He was with The University of Hong Kong from 1998 to 2000, and a Visiting Scholar with the University of Southern California from 1999 to 2000. He is currently a Cheung Kung Scholars Chair Professor of computer science and engineering with HUST. He has co-authored 22 books and published over 700 research papers. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security. He is a fellow of CCF and a member of the ACM. He received the Excellent Youth Award from the National Science Foundation of China in 2001. In 1996, he received the German Academic Exchange Service Fellowship to visit the Technical University of Chemnitz, Germany.

XUANHUA SHI (SM'17) received the Ph.D. degree in computer engineering from the Huazhong University of Science and Technology, China, in 2005. From 2006, he held an INRIA post-doctoral position with the PARIS Team at Rennes for one year. His current research interests focus on the cloud computing and big data processing. He published over 90 peer-reviewed publications, received research support from a variety of governmental and industrial organizations, such as the National Science Foundation of China, Ministry of Science and Technology, Ministry of Education, and European Union. He is a Senior Member of CCF and a member of ACM. He has chaired several conferences and workshops, and served on technical program committees of numerous international conferences.