Semi-synchronized Non-blocking Concurrent Kernel Cruising

Donghai Tian, Qiang Zeng, Dinghao Wu, *Member, IEEE,*Peng Liu, *Member, IEEE,* Changzhen Hu

Abstract—Kernel heap buffer overflow vulnerabilities have been exposed for decades, but there are few practical countermeasure that can be applied to OS kernels. Previous solutions either suffer from high performance overhead or compatibility problems with mainstream kernels and hardware. In this paper, we present KRUISER, a concurrent kernel heap buffer overflow monitor. Unlike conventional methods, the security enforcement of which is usually inlined into the kernel execution, Kruiser migrates security enforcement from the kernel's normal execution to a concurrent monitor process, leveraging the increasingly popular multi-core architectures. To reduce the synchronization overhead between the monitor process and the running kernel, we design a novel semi-synchronized non-blocking monitoring algorithm, which enables efficient runtime detection on live memory without incurring false positives. To prevent the monitor process from being tampered and provide guaranteed performance isolation, we utilize the virtualization technology to run the monitor process out of the monitored VM, while heap memory allocation information is collected inside the monitored VM in a secure and efficient way. The hybrid VM monitoring technique combined with the secure canary that cannot be counterfeited by attackers provides guaranteed overflow detection with high efficiency. We have implemented a prototype of KRUISER based on Linux and the Xen/KVM hypervisor. The evaluation shows that Kruiser can detect realistic kernel heap buffer overflow attacks in cloud environment effectively with minimal cost.

Index Terms—kernel, virtualization, heap buffer overflow, semi-synchronized, non-blocking, concurrent monitoring.

1 INTRODUCTION

Buffer overflows have been comprehensively studied for many years, but they remain as one of the most severe vulnerabilities. According to the National Vulnerability Database, 2425 buffer overflow vulnerabilities were reported in the recent three years, and more than 70% of them were marked as high severity [1].

Buffer overflows can be roughly divided into two categories: stack-based buffer overflows and heap-based buffer overflows. Both of them exist in not only user space but also kernel space. Compared with user-space buffer overflows, kernel-space buffer overflow vulnerabilities are more severe considering that if once such a vulnerability is exploited, attackers could override any kernel-level protection mechanism. Recently, more and more real buffer overflow exploits have been released in modern operating systems including Linux [2], OpenBSD [3] and Windows systems (e.g., Windows 7 [4] and Windows 10 [5]).

Many effective countermeasures against stack-based buffer overflows have been proposed, some of which, such as StackGuard [6] and ProPolice [7], have been widely deployed in compilers and commodity OSes. On the other hand, practical countermeasures against heap-based buffer overflows are few, especially in the kernel space. To our knowledge, there are no practical mechanisms that have been widely deployed to detectkernel space heap buffer overflows. Previous methods suffer from two major limitations: (1) some of them perform detection before each buffer write operation [8], [9], [10], [11], [12], which inevitably introduces considerable performance overhead. This kind of inlined security enforcement can heavily delay the monitored process when the monitored operations become intense; (2) some approaches do not check heap buffer overflows until a buffer is deallocated [13], [14], so that the detection occasions entirely depend on the control flow, which may allow a large time window for attackers to compromise the system. Other approaches [15], [16] either depend on special hardware or require the operating system to be ported to a new architecture, which are not practical for wide deployment.

In this paper, we present Kruiser, a concurrent kernel heap overflow monitor. Unlike previous solutions, Kruiser utilizes the commodity hardware to achieve highly efficient monitoring with minimal changes to the existing OS kernel. Our high-level idea is consistent with the canary checking methods, which firstly places canaries into heap buffers and then check their integrity. When a canary is found to be tampered, an overflow is detected.

Different from conventional canary-based methods that are enforced by the kernel inline code, we make use of a separate process, which runs concurrently with the OS kernel to keep checking the canaries. To address the concurrency issues between the monitor process and OS kernel, we design an efficient data structure that is used to collect canary location information. Based on this data structure, we propose a novel semi-synchronized algorithm, by which the heap allocator does not need to be fully synchronized while the monitor process is able to check heap canaries continuously. The monitor process is constantly checking kernel heap buffer overflows in an infinite loop. We call this technique *kernel cruising*. Our semi-synchronized cruising algorithm is non-blocking. The kernel execution is not blocked by monitoring, and monitoring is not blocked by the kernel execution. Thus the performance overhead and the impacts on kernel execution characteristics are very small on a multi-core architecture.

- *Donghai Tian is with School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China. E-mail: dhai@bit.edu.cn.*
- *Qiang Zeng is with Department of Computer Science and Engineering, University of South Carolina, Columbia, SC 29201.*
- *Dinghao Wu, and Peng Liu are with College of Information Sciences and Technology, The Pennsylvania State University, PA 16802.*
- *Changzhen Hu is with School of Software, Beijing Institute of Technology, Beijing 100081, China.*

This paper is an extension of the work originally reported in Proceedings of the 19th Network and Distributed System Security Symposium (NDSS 2012), San Diego, California, February 5--8, 2012.

Moreover, we apply a cryptography method to generate different canaries for different kernel buffers. In this way, it is difficult for attackers to recover canaries after launching the kernel heap overflows.

We have explored kernel heap management design properties to collect heap buffer information at page level instead of individual buffers. A conventional approach is to maintain the collection of canary addresses of live buffers in a dynamic data structure, which requires hooking per buffer allocation and deallocation. Instead of interposing per heap buffer operation, we explore the characteristics of kernel heap management and hook the much less frequent operations that switch pages into and out of the heap page pool, which enables us to use a fix-sized static data structure to store the metadata describing all the canary locations. Compared to using a dynamic data structure, our approach avoids the overhead of data structure growth and shrink; more importantly, it reduces overhead and complexity of the synchronization between the monitor process and the canary collecting code.

To provide performance isolation and prevent the monitor process from being compromised by attackers, we take advantage of virtualization to deploy the monitor process in a secure execution environment. Kruiser employs the Direct Memory Mapping technique, by which the monitor process can perform frequent memory introspection efficiently. On the other hand, the buffer address information is collected inside the VM to avoid costly hypervisor calls; Secure In-VM (SIM) [17] approach is adapted to protect the metadata from attackers.

In summary, we make the following contributions:

- **Semi-synchronized concurrent monitoring:** We propose a novel non-blocking concurrent monitoring algorithm, in which neither the monitor process nor the monitored process needs to be fully synchronized to eliminate concurrency issues such as race conditions; the monitor keeps checking live kernel memory without incurring false positives. We call this *semi-synchronized*. Concurrent monitoring leverages more and more popular multicore architectures and thus the performance overhead is lower compared to inline security enforcement.
- **Kernel cruising:** The novel cruising idea has been recently explored [18], [19]. It is nontrivial to apply this to kernel heap cruising.
- **Page-level buffer region vs. individual buffers:** We explore specific kernel heap management design properties to keep metadata at page level instead of individual buffer level. This enables very efficient heap buffer metadata bookkeeping via a static fixed-size array instead of dynamic data structures and thus reduces the performance overhead dramatically.
- **Out-of-VM monitoring plus In-VM interposition:** The isolated monitor process along with direct memory mapping through virtualization is applied to achieve efficient out-of-the-box monitoring. Moreover, we apply the SIM framework to protect the In-VM metadata collection. The hybrid VM monitoring scheme provides a secure and efficient monitoring in cloud environment.
- **Secure canary:** Unlike conventional canaries, which can be inferred and then counterfeited based on other canary values, we proposed the conception of *secure canary* and provided an efficient solution. Secure canaries along with the hybrid VM monitoring scheme guarantee the detection of buffer overflow attacks.

We have implemented a prototype of Kruiser based on Linux and the Xen/KVM hypervisor. The effectiveness of Kruiser has been evaluated and the experiments show that Kruiser can detect kernel heap overflows effectively. With respect to performance and scalability, our kernel cruising approach is practical—it imposes negligible performance overhead on SPEC CPU2006, and the throughput slowdown on Apache is 7.9% on average.

2 Problem Statement

Modern virtualization technique plays an important role in cloud computing. As more and more software services are deployed into virtualization environments, virtualization (or cloud) security becomes very important. Many researches focus on protecting the security of applications running in the virtualization (or cloud) environments. On the other hand, the OS kernel security is critical. If once the OS kernel is compromised, attackers could abuse the highest privileges to achieve their malicious purposes. Among all the kernel threats, the kernel heap buffer overflow is one of the most dangerous threats considering that it can lead to arbitrary code execution in the kernel space. The major reason for this threat is that the OS kernel component (or kernel module) does not perform boundary check when it accesses the kernel heap buffer. To our knowledge, the kernel heap security is not well addressed due to its complexity. How to provide a kernel heap protection mechanism that can efficiently detect a kernel heap buffer overflow in the virtualization (or cloud) environment is an important research problem.

3 Threat Model

This paper is focused on monitoring kernel heap buffer overflows. Other security issues, such as memory content disclosure or shellcode injection by exploiting format string vulnerabilities, are not in the scope of this paper. We assume the goal of an attacker X is to compromise the kernel of a VM; then he can do anything the kernel can do. Attacker X can launch arbitrary attacks against the kernel, but we assume that the kernel has not been compromised by other attacks launched by X before a heap overflow attack succeeds. Otherwise, he had already achieved his goal. Once the attacker X has compromised the kernel using heap buffer overflows, we assume X can do anything the kernel is authorized to do regarding memory read/write, OS control flow altering, etc. Since this work relies on the virtualization technology to monitor the kernel heap, we assume the underlying hypervisor is trusted. We could leverage previous research (e.g., HyperSafe [20], HyperSentry [21] and HyperCheck [22]) to protect the hypervisor security. Moreover, our trusted computing base includes a trusted VM where the monitor resides. This monitoring VM is special, and it is not supposed to run any other application. Thanks to the virtualization isolation, it is very difficult for attackers to subvert the monitoring VM that singly runs a user program. To further reduce the attack surface, the monitoring VM does not provide any service outside.

4 Background

SLAB allocator. Linux adopts the *slab* allocator [23][1] for kernel heap management. It uses *caches* to organize and manage heap

1. Linux can also use the SLUB/SLOB allocator to manage kernel heap. In this paper, we focus on the SLAB allocator for its fast performance on a multiprocessor system.

buffer objects. There are two types of caches in kernel heap, namely *general caches* and *specific caches*. General caches are mainly used to serve kmalloc calls requesting heap buffers of various sizes, while each specific cache is used to allocate objects of a specific kernel data structure, such as task_struct. A cache consists of one or more *slabs*, each of which occupies one or more physically contiguous pages and contains objects of the same type. The metadata of a slab describes the object arrangement and allocation status within the slab page (or pages).

Whenever a kernel component needs a kernel object, the *slab* allocator will return a prepared one if the slab is already constructed and has a free object. Otherwise, the allocator will carry out the slab construction and initialize the associated objects. When a kernel object is freed, it doesn't get destructed, but simply returned to its *slab*.

Canary-based Detection of Heap Overflows. Robertson et al. [13] first present the canary-based approach for heap overflow detection. The basic idea of this approach is to hook the allocator for adding 4-byte canary at the end of buffer during the dynamic allocation. When the dynamic buffer is freed, the allocator will check the canary integrity for the heap overflow detection. This approach is not new, but it is non-trivial to be applied for the OS kernel.

5 Challenges and Solutions

5.1 Overview

Based on the canary-based method, we propose a concurrent heap monitoring method for the OS kernel. The key idea is to generate a separate monitor process for checking the canaries. By instrumenting the slab allocator, our system attaches one canary word at the end of each heap buffer. The monitor process keeps scanning, or *cruising*, the canaries to detect buffer overflows.

5.2 Challenges

C1. Synchronization. Since the monitor process checks heap memory which is shared and modified by other processes, synchronization is vital to ensure the monitor process to locate and check live buffers reliably without incurring false positives.

Lock-based approach: A straightforward approach is to walk along the existing kernel data structures used to manage heap memory, which is usually accessed in a lock-based manner. This requires the monitor process to follow the locking discipline. When the lock is held by the monitor process, other processes may be blocked. On the other hand, the monitor process needs to acquire the lock to proceed. Both the kernel performance and monitoring effect will be affected using the lock-based approach. Another approach is to collect canary addresses in a separate dynamic data structure such as a hash table. By hooking per buffer allocation and deallocation, the canary address is inserted into and removed from the hash table, respectively. Nevertheless, it still does not reduce but migrate the lock contention, since the monitor process and other processes updating the hash table are synchronized using locks.

Lock-free approach: Scanning volatile memory regions without acquiring locks is hazardous [19], which usually needs to suspend the system to double check when an anomaly is detected. The whole system pause is not desirable and sometimes unacceptable. Another approach is to maintain the collection of canary addresses in a lock-free data structure. All processes update and access the data structure in a non-blocking manner. However, the contention between accessing processes may still lead to high overhead.

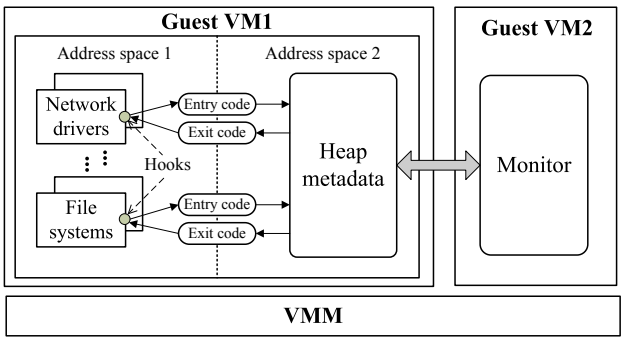

Fig. 1. Overview of Kruiser.

C2. Self-protection and canary counterfeit. As a countermeasure against buffer overflow attacks, our component can become an attack target itself. We rely on a monitor process that keeps checking—that is, *cruising*—the kernel heap integrity. After the system is compromised by exploiting the buffer overflow vulnerabilities, attackers may try to kill the monitor process to disable the detection completely. Attackers can also tamper or manipulate the data structure needed by our component to mislead or evade the detection. Moreover, attackers may try to recover the canary after corrupting it.

C3. Compatibility. Kernel heap management is among the most important components in OS kernels, whose data structures and algorithms are generally well designed and implemented for efficiency. Thus, the concurrent heap monitoring should not introduce much modification for heap management. Moreover, the solution should be compatible with mainstream systems as well as hardware.

5.3 Solutions

The Fig. 1 presents the overview of our solution. Specifically, the monitor process is running as a user-space program inside a separate VM (i.e., Guest VM2) for self-protection. The heap buffer metadata and hooking code are kept in the monitored VM (i.e., Guest VM1) to achieve efficient buffer information collection. Moreover, the heap metadata are stored in the fixed size memory area to facilitate Out-of-VM monitoring. The monitor accesses the inter-VM heap metadata via an efficient technique called direct memory mapping. To achieve a concurrent monitoring, the monitor process needs to locate and access the canaries reliably and efficiently, while the monitored system allocates and deallocates the buffers and heap pages continuously.

To address the synchronization challenge (C1), We explore the characteristics of kernel heap management, and propose to interpose heap page allocation and deallocation, by which we maintain concise metadata describing canary locations in a separate efficient data structure. Compared with interposing per buffer allocation and deallocation, the interposition is lightweight and the resultant overhead is much lower. The per page metadata is concise, which enables us to use a fix-sized static data structure to store it. Compared with using a concurrent dynamic data structure to collect canary addresses, the contention due to synchronizing data structure growth and shrink and the overhead due to data structure maintenance (node allocation and deallocation) are completely eliminated. More importantly, as the monitor

process traverses our own data structure rather than relying on existing kernel data structures, it is more flexible to design the synchronization algorithm, i.e. the monitor process does not need to follow the synchronization discipline imposed by the kernel data structure. Therefore, we are able to design a highly efficient semi-synchronized non-blocking algorithm, which enables the monitor process to constantly check the live memory of the monitored kernel without incurring false positives.

To address the self-protection and canary counterfeit challenge (C2), we apply the virtualization technology to deploy the monitor process into a trusted environment (Fig. 1). To ensure the same efficiency as in-the-box monitoring, we introduce the Direct Memory Mapping (DMM) technique, which allows the monitor process to access the monitored OS memory efficiently. To protect the heap metadata and interposition code from being compromised by attackers, we apply the SIM [17] framework, which enables the data and code to be protected safely and efficiently inside the monitored VM. As shown in Fig. 1, by utilizing the VMM, we introduce two separate address spaces in the monitored VM, and address space 2 is used to place the heap metadata and interposition code. The entry code and exit code are the only ways to transfer execution between the two address spaces so that the metadata can be updated. Canaries are generated applying efficient cryptography, such that once a canary is corrupted, it is difficult for attackers to infer and then recover the canary value.

To address the compatibility challenges (C3), we made minimal changes to the existing kernel heap management based on the commodity hardware. Specifically, we hook the allocation/deallocation that adds/removes pages into/from the heap page pool to update the corresponding heap metadata in our data structure, so that kernel heap buffer allocation algorithms are not changed. On the other hand, the major monitor component is located out of the monitored kernel leveraging the popular VMM platform, which is widely used in cloud computing nowadays.

6 Kernel Cruising

In this section, we present the semi-synchronized non-blocking kernel cruising algorithm. We introduce the data structure used in the algorithm in Section 6.1. We discuss potential race conditions in Section 6.2 and describe our algorithm in Section 6.3.

6.1 Page Identity Array

Kernels usually maintain heap metadata in dynamic data structures. For example, Linux kernel uses a set of lock-based lists to describe the heap page pool. It is tempting to walk along the existing data structures to check heap buffers. This way the concurrent monitor process has to follow the locking discipline, which would introduce intense lock contention. Another concurrent approach, as used in kernel memory mapping and data analysis for kernel integrity checking [19], is to check without acquiring locks and freeze the monitored VM for double-check to avoid false positives, which may require suspending the VM frequently in our case.

Instead of relying on kernel-specific data structures, we maintain a separate structure called Page Identity Array (PIA). Its basic form is a static array data structure with each entry recording the *identity* of a page frame. A variety of page identity information can be of interest, such as per page signature, access control, accounting and auditing data. With regard to concurrent heap monitoring, a PIA entry records whether a page frame is used for heap memory, and if so, the metadata that is used to locate canaries within the page. The first entry corresponds the first page frame, and so forth. Since the kernel memory address space is fixed, the size of PIA structure can be predetermined. This way we only need to hook functions that add pages into the heap page pool and that remove pages from it, updating metadata in the corresponding entries. The monitor traverses the PIA structure and check canaries according to the stored metadata. Compared to interposing per buffer allocation and deallocation and collecting canary addresses in a dynamic data structure, the overhead due to function hooking and data structure maintenance is largely reduced. We postpone details about metadata and memory overhead analysis in Section 7.

The idea of using a fixed-size data structure is due to the insight into kernel heap management. We assume that a kernel page, if used for heap memory, is divided into buffer objects of equal size and that all the buffers in this page are arranged as an array, which is true in most commodity systems. Given a heap page and its initial buffer object address and size, the monitor process can locate all the buffers within this page, such that the metadata stored in each PIA entry can be small. Before a process (or a kernel thread)[2] adds a page into the heap page pool, the canaries within the page are initialized and the corresponding PIA entry is updated. By scanning the canaries within each page, the monitor process detects buffer overflows. Although some buffer objects are not allocated and some canary checking may be not necessary, the simple read operations do not introduce much overhead.

6.2 Race conditions

Exploring the characteristics of kernel heap management, we proposed the static PIA structure, which avoids heap monitoring from relying on kernel-specific heap data structure and supports highly efficient random access. Nevertheless, synchronization between the monitor process and processes updating page identities is still an issue. For example, when the monitor process reads an entry, another process may be updating it. Without synchronization, the consistency of PIA entries cannot be ensured, which implies the monitor process cannot retrieve heap buffers reliably.

Before we present the kernel heap cruising algorithm, we first discuss the potential race conditions for sharing the PIA structure, which motivate our semi-synchronized design in Section 6.3. Three categories of processes need to access the PIA structure: the monitor process, processes updating PIA entries when pages are added into and removed from the pool, respectively. When multiple processes access the PIA structure, a variety of race conditions can occur, some of which are subtle.

Non-atomic entry write: As updating a PIA entry is not atomic, a race condition occurs if we allow multiple processes to modify the same entry simultaneously, which would corrupt the entry. Lock-based synchronization is simple, but it incurs high performance overhead and blocks heap operations.

Non-atomic entry read: When the monitor process is reading a PIA entry, another process may be updating it. However, as the read and update of an entry are not atomic, the monitor process may read inconsistent entry values.

Time of check to time of use (TOCTTOU): For a given entry if the corresponding page is in the heap pool, the monitor process checks canaries within that page, during which, however, the page

2. In this paper we will use the two terms interchangeably.

```
1 //Add a page into the heap page pool
2 AddPage(page){
3     ...
4     /* Inside critical section */
5     Initialize all the canaries within the page
6     Update the metadata in PIA[page];
7     smp_wmb(); // This write memory barrier
          enforces a store ordering
8     PIA[page].version++;
9     ...
10 }
11
12 //Remove a page out of the heap page pool
13 RemovePage(page){
14     ...
15     /* Inside critical section */
16     for (each canary within the page)
17         if (the canary is tampered)
18             alarm(); // A Buffer overflow is
                   detected
19    PIA[page].version++;
20     ...
21 }
22
23 Monitor(){
24     uint ver1, ver2;
25     for (int page = 0; page < ENTRY_NUMBER; page++)
           {
26         ver1 = PIA[page].version;
27         if (!(ver1 % 2))
28             continue; // Bypass non-heap page
29
30         smp_rmb(); // This read memory barrier
               enforces a load ordering
31         Read the metadata stored in PIA[page];
32         smp_rmb();
33         ver2 = PIA[page].version;
34         if (ver1 != ver2)
35             continue; // Metadata was updated
                   during the read
36
37         for (each canary within the page){
38             if (the canary is tampered)
39                 DoubleCheckOnTamper(page, ver1);
40         }
41     }
42 }
43
44 DoubleCheckOnTamper(page, ver){
45     uint ver_recheck = PIA[page].version;
46     if (ver_recheck != ver)
47         return; // The page was already removed/
               reused
48     alarm(); // A buffer overflow is detected
49 }
```

Fig. 2. Kruiser monitoring algorithm.

may be removed from the pool and used for other purposes, such that false alarms may be issued.

To avoid false alarms, it is tempting to double check whether the page has been removed from the heap page pool when a canary is detected tampered. Specifically, a flag field indicating whether the page is in the pool is contained in each entry. A process removing the page out of the heap page pool resets the flag; when a heap buffer corruption is detected, the monitor process double checks the flag to make sure the page is still in the pool. A buffer overflow is reported only when a canary is tampered and the flag in the PIA entry is not reset. However, it cannot avoid the *ABA* hazard as discussed below.

ABA hazard: An ABA hazard occurs when one process reads a value A from some position, and then needs to make sure the position is not updated since last access by reading it again and comparing the second read value with A. However, between the two reads, other processes may have updated the position from value A to B then back to A. In our case, it may lead to an ABA hazard if the monitor process intends to determine whether the entry has been updated by reading the flag twice, considering that other processes may have removed the page from the heap page pool and then added it back between the two reads, such that the idea of double-checking the flag can still lead to false alarms due to ABA hazards.

Compared to the idea of walking along existing kernel data structures, we apparently have conquered nothing except migrating the synchronization problems to the PIA structure. However, as presented below, we propose a semi-synchronized algorithm based on PIA to resolve all the problems without incurring false positives or high overhead.

6.3 Semi-synchronized Non-blocking Cruising

We propose an efficient semi-synchronized non-blocking kernel cruising algorithm, as shown in Fig. 2, that works with the PIA structure. It resolves the concerns of race conditions without introducing complex synchronization mechanisms, such as fine-grained locks and intricate lock-free data structures.

We add an unsigned integer field version in each entry, which records the "version" of the corresponding page. It is initialized to be an even number when the corresponding page is not in the heap page pool. Whenever a page is added into or removed from the pool, its corresponding version number is incremented by one, so that an odd version number indicates a heap page, and an even number indicates a non-heap page. Because the size of the version field is one word, the read and write of a version value is atomic, which is critical for the correctness of our algorithm.

Avoid Concurrent Entry Updates: The kernel commonly has its own synchronization mechanisms to prevent one page frame from being manipulated for inconsistent purposes at the same time. For example, Linux functions kmem_getpages and kmem_freepages, which add page frames into and remove them from the heap page pool, respectively, operate on page frame in a critical section with lock protection. These two functions correspond to AddPage and RemovePage in Fig. 2, respectively. The PIA entry update operations can be put into the critical section of these two functions; it is thus ensured that two processes cannot update the same entry simultaneously. By leveraging the existing synchronization mechanisms in kernel to maintain the PIA entries, the additional overhead is minimal since updating metadata in a PIA entry is fast. As long as the kernel prevents one page frame from being manipulated by two processes simultaneously, there should be synchronization mechanisms serving for this purpose, so the "free-ride" is widely available.

Avoid Using Inconsistent Entry Value: Instead of preventing the monitor process from reading inconsistent entry value, we allow it to occur. However, we use a double-check algorithm to detect potential inconsistency and avoid using inconsistent values. We read the version field in an entry first (Line 26), and then retrieve other entry fields followed by another read of the version field (Line 33). The page is to be scanned if and only if the two reads of the version field retrieve identical odd version numbers. Here we assume the wraparound of the version value does not occur between the two reads. Considering that page frame switch in and out of the kernel heap pool is infrequent, it very unlikely that the

version number wraps around a 32-bit unsigned integer between the two reads.

Specifically, assume there is a non-heap page frame and the AddPage function adds it into the heap page pool. In its critical section it first updates the metadata and then the version number (Line 8) in the corresponding page entry, such that if the monitor process reads the version number of the entry being updated and the read is before the version number update (Line 8), it will retrieve an even number, which indicate a non-heap page. The monitor process will bypass this page (Line 27) according to our algorithm. A write memory barrier (Line 7) is inserted before the version number update, which preserves an observable update order. It is a convention to assume a sequential consistency memory model in the parallel computing literature when describing a concurrent algorithm; however, the observable update sequence [24] is vital to the correctness of our algorithm, so we point it out explicitly.

The version number is not incremented until RemovePage removes the page from the pool. It does not need write memory barriers around the version update because the enter and exit of a critical section imply a full memory barrier, respectively. Therefore, as long as the two reads of the version field retrieves identical odd values, the retrieved metadata values are consistent. Two read memory barriers (Line 30 and 32) are inserted into the Monitor function, such that an observable load ordering is enforced among the reads of the version number and metadata. But note that the read and write memory barriers are not needed on x86 and AMD64 platforms [25], as they already preserve the loads and stores orders we need.

Identify TOCTTOU and ABA Hazards: Without locks or other synchronization primitives, it is difficult to avoid TOCTTOU and ABA hazards. Rather than avoiding the hazards, the algorithm takes a different approach to recognizing potential hazards to avoid false alarms. When a canary is found changed, the monitor process does not report an overflow immediately. Instead, it makes sure the page being checked has not ever been removed out, which is indicated by the version number again. As long as the version number does not change compared to the last read (Line 46), it can be determined that the page has persisted as a heap page; in this situation, if a canary is found corrupted, a buffer overflow is reported without concerns of false positives.

Correctness Analysis: As shown in the Fig. 2, our monitor first checks the version field of the PIA entry. If it is an even number, the monitor will skip scanning canaries within this memory page. After the monitor just finishes these operations, the kernel may have initialized the canaries and PIA metadata for that page. As a result, the newly allocated page has to wait for a cruising cycle (i.e., the time to scan all the PIA entries) to get checked. Our evaluations show that the maximum detection latency is less than 40 milliseconds. Therefore, the time window for attackers to recover the corrupted canary is very small. In addition, our secure canary mechanism (section 7.4) guarantees that the corrupted canary is extremely difficult to get recovered.

On the other hand, if the version field is an odd number, the monitor will read the metadata stored in the PIA entry. In the meanwhile, the kernel may be in the procedure of removing the corresponding page from the heap page pool. Consequently, we need to handle two situations shown in the Fig. 3.

For situation (1), the monitor finishes reading the PIA entry and rechecking the version field before its value gets updated by the kernel. Since the PIA metadata is not changed for the consistent version number, the monitor will use this metadata to check the canaries. Before the checking is finished, the associated memory page may be removed or reused. As a result, the monitor may find the canary is tampered. To eliminate this false positive, the monitor will recheck the version field again. If the value is changed, it is a false alarm, Otherwise, it indicates a real buffer overflow.

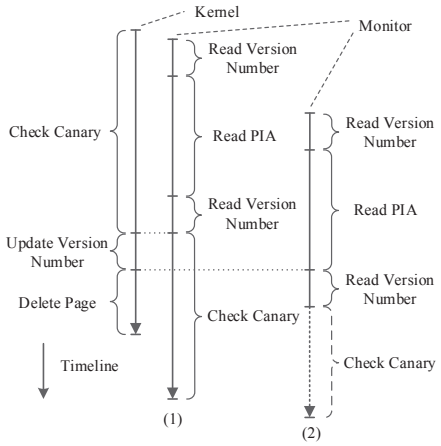

Fig. 3. Different situations for the monitor to read the version number when it is updated by the kernel.

For situation (2), the monitor rechecks the version field of the PIA entry after its value is already updated by the kernel. Since the version number is changed, the monitor will skip scanning the corresponding memory page. In this way, the unnecessary check can be avoided.

The non-blocking algorithm is constructed using simple reads, writes, and memory barriers without introducing complicated and expensive synchronization mechanisms. The monitoring is *wait-free* as it guarantees progress in a finite steps of its own execution; i.e., it is non-blocking. The monitor process reads version numbers to determine its control flow, so it is lightly synchronized, while other processes manipulating heap pages make progress without being synchronized or blocked by the monitor process. In other words, the synchronization is one-way. That is why we call it a *semi-synchronized non-blocking cruising*. On PIA entries, write-write is synchronized with a free-ride from the existing kernel functions, while read-write is not synchronized. It resolves the concern of a variety of subtle race conditions without the need of freezing the entire system for recheck. It does not have false positives and enables efficient concurrent heap monitoring.

7 System Design and Implementation

7.1 Architecture

The architecture, as shown in Fig. 4, can be divided into three parts: VMM, Dom0 VM, and DomU VM. The Monitor Process in Dom0 VM executing Monitor (Fig. 2) in an infinite loop to monitor the kernel of DomU VM. A tiny component, namely Memory Mapper, inside the VMM is used to map the kernel memory of the monitored VM to the monitor process. For this purpose, Memory Mapper needs to manipulate the user page table of the monitor process, which is detailed in Section 7.2. The custom driver in Dom0 VM is used to assist the monitor process to release extra memory during the memory mapping.

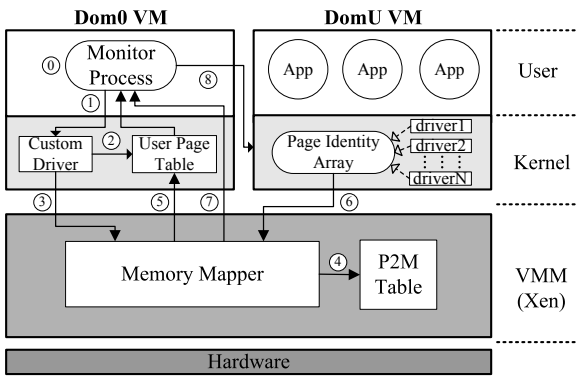

Fig. 4. Kruiser Architecture. The numbers in the small circle indicate Kruiser's work flow.

The Page Identity Array and the interposition code reside in the kernel space of DomU VM. By looking up this array, the Monitor Process can get all the heap canary location information. To update the PIA, the interposition code has to hook the slab allocations and deallocations. For the PIA and interposion code protection, Section 7.3 presents the solution.

The out-of-VM monitoring ensures performance isolation and secureness, but usually leads to high overhead. The in-VM information collection provides native code execution and memory access environments, but may be vulnerable to attacks. By addressing the problems, we combine the two schemes as a hybrid solution to provide a secure and efficient monitoring.

7.2 Direct Memory Mapping

To achieve an out-of-the-box monitoring, a conventional method is to run a monitor process in a trusted VM and perform virtual machine introspection (VMI) via the underlying VMM. However, frequent memory introspection would incur high performance overhead. Each such operation requires VMM to walk the monitored VM's page table and map the target machine frames to be accessible from the monitor process. To avoid this problem, we introduce Direct Memory Mapping (DMM), by which the monitor process can perform frequent memory introspection with only one-time involvement of the VMM. Fig. 5 shows the basic idea of DMM. By manipulating the underlying memory mapping, the VMM can ensure the monitor to access the kernel memory of the monitored OS directly. The procedure of DMM can be divided into three stages.

First, the Monitor Process allocates a chunk of memory whose size is determined by the maximum number of memory pages used for DomU VM's kernel heap (⓪ in Fig. 4). For Linux, the kernel heap only resides in the memory pages that are directly mapped by the OS kernel. In a 32-bit kernel, the maximum size is 896MB even if the machine physical memory size is bigger than 896MB. In a 64-bit kernel, the maximum size can be as large as 64TB. The goal of this stage is to create a contiguous range of virtual addresses. By properly manipulating the page table entries (PTEs), the VMM enables the monitor process to access the memory of the target OS kernel within the monitor's virtual address space. However, due to the demand paging mechanism adopted by Linux, the PTEs are not actually established when the host virtual addresses (HVAs) are created. Therefore, we need to access all the created HVAs to trigger the creation of PTEs before operating on them.

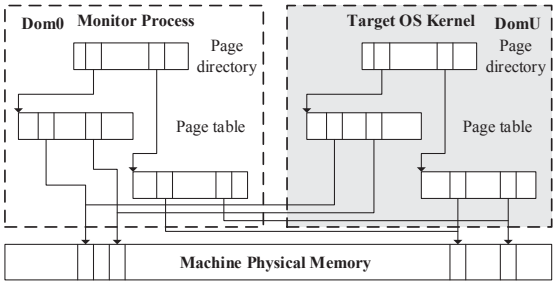

Fig. 5. The direct memory mapping mechanism.

Second, the Monitor Process notifies the Custom Driver to reclaim the newly allocated pages (①) with the PTEs retained. This is necessary because the Monitor Process only needs the new HVAs and the corresponding PTEs but does not use the allocated pages; returning these pages back can save a lot of physical memory. Specifically, this stage consists of four steps. 1) The Custom Driver first walks the page table of the Monitor Process to identify the PTEs for the memory chunk allocated in the first stage (②). 2) Then, with these identified PTEs, the Custom Driver searches for the corresponding page descriptors used by the page frame management. 3) After that, the Custom Driver clears the relevant flags in these page descriptors (e.g., active flag), and resets their reference counters, map counters as well as other related information. 4) Finally, the Custom Driver invokes the API of the buddy system (i.e., __free_page()) to release the page frames.

Third, after the Custom Driver finishes reclaiming pages, it informs the Memory Mapper to perform DMM for the Monitor Process (③). By looking up the DomU's physical-to-machine (P2M) table (④), the Memory collects all the machine frame numbers (MFNs) of the DomU. With the mapping information, the Memory Mapper updates the PTEs of the Monitor Process accordingly. Specifically, given the newly allocated virtual address range, the Memory Mapper walks the User Page Table to find the corresponding PTEs (⑤), whose page frame numbers are then changed to the MFNs that are collected from the P2M table. In this way, the Monitor Process can access the entire kernel of the target OS with its own page table.

Once the Page Identity Array is allocated and initialized in DomU VM, it invokes a hypercall to notify the underlying VMM (⑥), which then informs the monitor process to begin cruising over the kernel heap (⑦)(⑧).

Reducing TLB Pressure. As the memory area that the Monitor Process accesses may be large when a lot of kernel slabs are produced, the kernel cruising may incur high TLB pressure. To address this problem, we exploit the page size extension that is supported by commodity microprocessors. Specifically, we set the Page Size flag in the page directory entries, enabling the size of page frames to be 2MB instead of 4KB (the page frame will be 4MB in size if it is in None-PAE mode). Note that to this end we also need the hypervisor to support large pages. Fortunately, Xen (with PAE enabled) mainly uses 2MB super pages to allocate memory for guest VMs. On the other hand, to ensure the large page to work properly, we require the starting virtual address allocated for the monitor process should be 2MB-aligned. To meet this requirement, the Monitor Process needs to allocate 2MB extra memory during the first stage, and then adjust the starting virtual address to be 2MB-aligned before performing DMM.

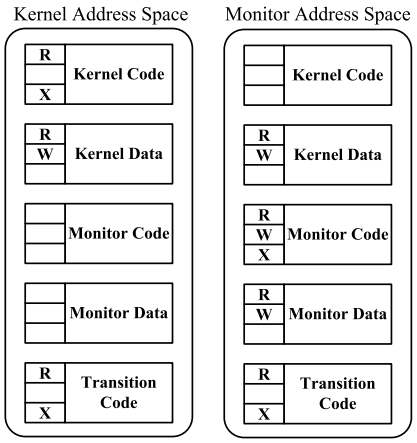

Fig. 6. Memory protections in the kernel and monitor address space.

Address Translation. As the kernel heap is only located in the memory region that is directly mapped into the kernel space, the Monitor Process can access the kernel heap of the target OS efficiently using simple address translation. Specifically, the monitor first figures out the physical addresses of kernel objects in the heap by subtracting the value of PAGE_OFFSET[3]. Then, it can get the private addresses by adding the physical addresses to the starting address of the newly allocated memory area. According to these user addresses, the Monitor Process can locate the corresponding kernel objects.

7.3 In-VM Protection

Since the PIA data structure (metadata) and the interposition code reside in the kernel space of DomU VM, attackers may manipulate them directly after exploiting buffer overflow vulnerabilities. To solve this problem, a conventional method is to move the data structure and code to be protected into the hypervisor or another trusted VM. However, it will incur significant performance overhead when the world switches between the hypervisor and the VM become frequent, especially for such fine-grained monitoring as in our case. Instead, we employ the SIM [17] framework, which enables a secure and efficient in-VM monitoring. Specifically, the hypervisor creates a separate protected address space inside DomU VM and puts the code and data to be protected in it, such that those memory regions are protected from the DomU VM kernel by the hypervisor, and the separate address space can only be entered and exited through specially constructed protected gates.

In our case, we need to move the interposition code added in the critical section of AddPage and RemovePage as well as the PIA data structure in Fig. 2 to the protected memory regions. To this end, we construct two shadow page tables (SPTs) specifying different access permissions for the kernel and the In-VM monitor part.[4] As shown in Fig. 6, within a kernel address space, a process is not allowed to access the monitor code and data regions, while the kernel code cannot be executed after a process switches to the monitor address space. To invoke the monitor's code in the kernel address space, the transition code is used to switch address spaces and is executable in both address spaces. The transition code modifies the *CR3* register, which contains the physical address of the root of the target shadow page table. By default, any change of *CR3* will result in a *VMExit*. Fortunately, a recent hardware feature allows us to change the *CR3* without being trapped to the hypervisor if its value is in the *CR3_TARGET_LIST*, which is maintained by the hypervisor.

3. In a 32-bit system, the offset value is 0xc0000000; in a 64-bit system, the offset value is 0xffff880000000000.

4. Note that the In-VM monitor part only includes the PIA and the interposition code and will be referred to as the *monitor* in this section for short, while the monitor process still runs out of the VM.

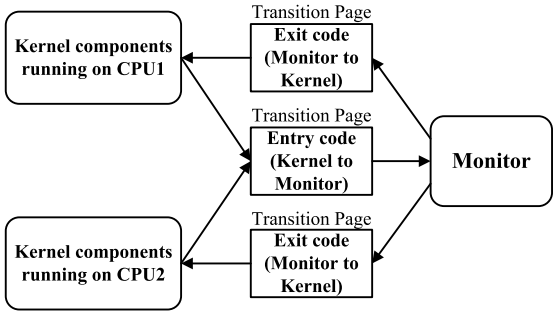

Fig. 7. Address space switching via transition pages in SMP.

Address Space Maintaining and Switching in SMP. Maintaining and switching shadow page tables in Symmetric Multi-Processing (SMP) involves two challenges: 1) The SPTs for the kernel address space and the monitor address space should get synchronized for correctness. 2) The transition code should determine the correct *CR3* target when switching back to the kernel address space.

To address the first challenge, a straightforward method is to modify the hypervisor so that any changes to one SPT used by the kernel (including allocating, changing and removing a SPT entry) always propagate to the other SPT used by the monitor. Unfortunately, this method will not only increase the extra synchronization performance overhead but also add the implementation complexity. Alternatively, our approach takes advantage of the fact that the monitor only needs to access the kernel heap (for placing canaries) and kernel stack (for accessing the arguments and storing local variables), whose memory areas are only located in the pages that are directly mapped by the OS kernel. Hence, by looking up the P2M table, we could build the memory mapping in the SPT used by the monitor with one-time effort, and then no synchronization is needed.

For the second challenge, one common solution is to apply a lock-based method so that only one single kernel component can use the transition page to switch the address space back and forth each time. However, doing so may affect the system concurrency. Moreover, given multiple SPTs that could be used at the same time in the SMP environments, this method may require to save and restore the current *CR3* value during the address space switching. Since the instructions, *MOV* from *CR3*, cause *VMExit* unconditionally, this approach could not reduce the performance overhead. Instead, we take a different approach, which is shown in Fig. 7. As the kernel components running on different CPUs share the same monitor address space after the switch, only one transition page is needed for the entry code.

On the other hand, considering the fact that different CPU may use different SPT, we are required to generate different transition pages for the exit code. Additionally, since each CPU may use different SPTs and switch them as necessary, our system must ensure that the *CR3* target where the exit code is going to transfer back equals to the address of shadow page directory being used by the current CPU prior to entering the monitor address space.

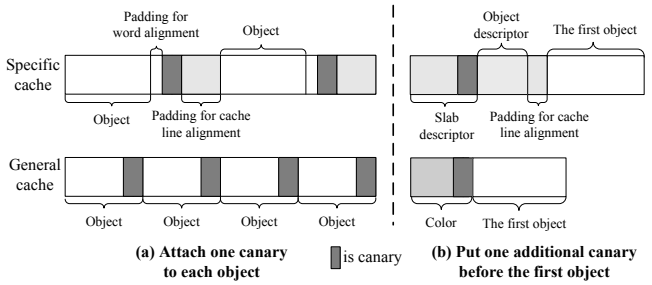

Fig. 8. Placing canaries into kernel objects.

To this end, we modify hypervisor to update the *CR3* target used in the associated exit code when a *per-CPU* switches one SPT to another. Accordingly, the hypervisor should also update the *CR3_TARGET_LIST*. To facilitate the monitor to select the corresponding transition page for transferring the execution back to the kernel component, we generate these pages according to the different *CPU ID*, which can be easily determined by the monitor (i.e., using the function smp_processor_id()).

Security Check. By invoking duplicate AddPage for the corrupted page, attackers can recover the canaries. To avoid this problem, we add one more check in the protected code to prevent pages with odd PIA version numbers from being added again. On the other hand, if attackers invoke RemovePage maliciously, the final round of canary checking in the function can detect overflows.

Additionally, we need to consider an attack scenario where the exploit installs something (e.g., rootkit) on the system and then reboots the kernel so that it would bypass our detection mechanism. To address this problem, we could utilize the hypervisor to mediate the reboot. Before the kernel is rebooted, we can pause the system for a while such that the monitor process will discover the corrupted canaries after scanning the entire kernel heap.

7.4 Placing Canaries

To detect underflows as well as overflows, it is straightforward to place two canaries surrounding each buffer. Actually, Linux with slab debug-enabled version has adopted this scheme to place canaries. Unfortunately, this method does not make kernel objects aligned in the first-level hardware cache, which may result in more cache misses. To overcome this limitation, we only use one canary instead of two canaries to surveil each kernel object. Since the same type of kernel objects are grouped together inside a slab, our approach can still detect the heap underflow attack occurred in one object (but not the first one in a slab) by checking the canary attached by the previous object.

As shown in Fig. 8(a), we apply two different ways to place the canary. For the specific caches, we first pad the objects to be word-aligned in size. Then, we add one word canary following the object. Finally, to ensure the object get L1 cache line aligned, we put some additional padding at the end of this object. On the other hand, as the objects in general caches have already got L1 cache line aligned in size, there is no need to change the form of these objects. Instead, we place a canary in the last word of each object. In addition, we hook the general object allocation function (i.e., kmalloc), and increase the original requested size by one word to hold the canary.

Although the scheme above works well to detect underflows (and overflows), it cannot deal with underflows occurred in the first object, as there is no canary preceding it. To tackle this issue, as shown in Fig. 8(b), we exploit the existing infrastructure to add a canary before the first object. Specifically, if the slab descriptor is located rightly before the first object, the canary is placed at the end of this slab descriptor; or if there is a slab color,[5] we put a canary in the last word of this color.

Secure Canary Generation. To set canary values for kernel objects, a practical solution should meet the two requirements: 1) after attackers have compromised the monitored kernel via buffer overflows, they cannot recover the corrupted canaries; 2) the canary generation and verification algorithms should be efficient so that they will not affect the system performance and detection latency. To satisfy these requirements, we employ a stream cipher (RC4 [26]) to generate canary values. For each slab, we first extract a random number from the entropy pool in Linux. Then, this random number is used as a key "stretched" by RC4 into a stream of bytes, the length of which is decided by the number of objects inside the slab. Finally, each 4 bytes of this stream is selected as a canary value for each object. On the other hand, to facilitate canary checking, we store the key (i.e., the random number) into the corresponding PIA entry for each slab.

Initially, the entropy pool is empty. As a result, extracting a random number from the entropy pool at that time will cause a problem. To address this issue, we modify the kernel to get the initial random numbers from the underlying hypervisor during the initialization. Specifically, the kernel first invokes a hypercall to request a random number. Then, the hypervisor returns the result by looking up Dom0 VM's entropy pool. Once the kernel is fully initialized, the hypervisor will shepherd the kernel to extract the random numbers from its own entropy pool for the next requests.

Guaranteed Detection. With the In-VM protection and secure canary generation, attackers can not hide their attacks in that 1) The In-VM protection prevent attackers from manipulating the PIA entries; 2) The canary generation based on the stream cipher guarantees the difficulty for attackers to recover the corrupted canaries within one cruising cycle. In addition, attackers cannot change the memory mapping between the monitor process and the monitored kernel in that the associated page table is maintained by another trusted VM (i.e., Dom0). Therefore, the attacks are bound to be detected within one cruising cycle after compromising the system, unless the attackers know the exact canary value to be corrupted beforehand, which usually implies the overread and overrun vulnerabilities overlap for exactly the same buffer area.

Moreover, to improve the difficulty for attackers to compromise and recover the canaries, we can change the linear scheme to scan the PIA entries. Specifically, we utilize the random number from the entropy pool as the first index for the PIA scan in each crusing. In this way, the detection latency could be less than one cruising cycle, thus reducing the kernel intrusion's survival time in probabilistic.

7.5 Locating Canaries

To locate and verify canaries in the Monitor Process, we hook the slab allocations and deallocations to store the metadata into the PIA entries, one of which is shown in Fig. 9. The mem field record the starting address of the first object within the slab. As each PIA entry corresponds to one physical page, we only need to

5. A slab color is a padding put in the beginning of each slab to optimize the hardware cache performance.

```
1 struct PIA_entry{
2     unsigned int version;
3     short mem;//the starting address of the first
          object
4     short slab_size;//the size of the slab
          descriptor
5     int obj_size;//the actual size used by each
          object
6     int buffer_size;//the whole size for each
          object
7     int number;//the number of objects in this slab
8     unsigned int key;//the key for canary
          verification
9 };
```

Fig. 9. PIA entry.

remember the last 12 bit of the address, which equals the offset within one page. For the obj_size field, we store the actual object size, including the size of padding for word alignment.

By adding the start address of one object and its actual object size, we can get the canary address. To acquire the start address of the next object, the PIA entry contains the buffer_size field, which refers to the whole object size after adding the canary as well as the padding for cache line alignment. The num field indicates the number of objects within a slab. To locate the canary that resides in the slab descriptor, we record the slab descriptor size in the slab_size field, which additionally includes the size of the object descriptor and the following padding. With the starting address of the first object subtracting the slab descriptor size, we get the starting address of the slab descriptor and then locate the canary, whose offset within the slab descriptor is predetermined. On the other hand, if the slab descriptor is kept off the slab, we set the value of the slab_size to zero. Accordingly, we employ a different method to locate the canary before the first object. In particular, we check whether the starting address of the first object is page-aligned, if not, it indicates there is a color placed in the front. Then, we can check the canary safely.

As introduced previously, kernel heap are managed in different slabs, one of which consists of one or more physically contiguous pages. Therefore, the slab that contains several pages should correspond to several entries in the PIA. In order to facilitate recording the slab canary information into PIA entries, we just use the first associated entry to store the whole information, and keep other associated entries empty.

It is worth mentioning that we utilize the page allocator to dynamically allocate kernel memory for the PIA data structure during the kernel's initialization. Basically, the total memory occupied by the PIA is determined by the number of pages in the heap. However, the proportion is unchanged even if all the physical memory are used by the kernel heap. Since each PIA entry has only 24 bytes in our implementation, the memory overhead is as low as 24/4096 for each memory page. For 1 GB physical memory, it just needs 6 MB memory to store the PIA. Furthermore, it is possible to reduce the size of the PIA entry by packing its fields.

7.6 Porting to 64-bit System

To apply our protection mechanism for the 64-bit system (i.e., x86-64 Linux 2.6.24), minimal changes of our implementation are needed. Since the size of 64-bit kernel heap is much larger than the 32-bit, the first change is to reset the kernel compile configuration. By doing so, the buddy system can allocate a large number of contiguous memory pages for storing the Page Identity Array (PIA). The second change is to put the PIA initialization code onto the location before the kernel cache is initialized. If not doing so, the kernel will crash. The third change is to adjust the data type so that the system can work correctly. For example, the size of a canary is set to a word size and a half word size in the 32-bit and 64-bit system respectively. To update the PIA entry, we need to obtain the physical page frame given the slab page allocation. In the 64-bit system, the kernel virtual to physical address conversion is a little different from that of the 32-bit system.

To monitor the 64-bit kernel heap, we utilize the recent KVM hypervisor (version 2.9.0). In KVM, the guest VM runs as a regular Linux process. To access the kernel heap of the guest VM, the monitor is deployed as a separate process and makes use of the memory sharing mechanism. Specifically, we first develop a custom kernel module to register a virtual device. Then, the monitor process opens this virtual device to map the physical memory region of the guest VM to its own address space. To obtain the host physical addresses allocated for the guest VM, we need to look up the GPA to HPA mapping maintained by the hypervisor. To reduce the engineering effort, we do not implement the in-VM protection mechanism for the 64-bit system.

8 Evaluation

To evaluate Kruiser, we developed a prototype of Kruiser based on 32-bit Linux and the Xen hypervisor (with PAE enabled), and conducted effectiveness tests and measured performance overhead. All the experiments were run on a Dell Precision Workstation with two 2.26GHz Intel Xeon quad-core processors and 6GB memory. The Xen hypervisor (with PAE enabled) version is 3.4.2. We used Ubuntu 8.04 (linux-2.6.24 with PAE enabled) as Dom0 system and Ubuntu 8.04 (linux-2.6.24 with PAE disabled) as DomU system (with HVM mode). For this DomU system, we allocated 1 GB memory and 4 VCPU. Moreover, we ported Kruiser to KVM for monitoring the kernel heap of the 64-bit Ubuntu 8.04 system (linux-2.6.24). The guest VM was allocated 2 GB memory and 4 VCPU.

8.1 Effectiveness

To test whether Kruiser can detect heap buffer overflows, we deliberately introduced four explicit vulnerabilities [27], [2] in the Linux kernel, and then exploited these bugs. In our first test, we modified the kernel function cms-ghdr_from_user_compat_to_kern, making it process some user-land data without sanitization, such that malicious users launch heap-based buffer overflow attacks via the sendmsg system call. For the second test, we loaded a vulnerable kernel module that is developed by ourselves. The function of this module is to use a dynamic general buffer to store certain data transferred from the user-land. However, the module does not perform boundary check when it stores the user data. In the third test, we also employed a loadable kernel module to export a bug in kernel space. Unlike the second test, we constructed a specific slab in this module, and allocated the last object in this slab to store certain user-land information [2]. As a result, this vulnerability enables attackers to overwrite a page next to the slab by transferring large size data into the kernel object. In the fourth test, we crafted a vulnerable kernel module and then exploited the kernel heap overflow vulnerability to mimic a realistic kernel attack for privilege escalation. To this end, we used the kmalloc function to allocate two kernel objects. One object was used to directly store the data transferred from

the user-land, while the other object contained a function pointer that is invoked periodically. After carefully preparing the heap layout where the two kernel objects became adjacent, we could carry out the overflow attack to overwrite the function pointer, and then execute the shellcode to raise the process credential. We then launched four types of heap-based buffer overflow attacks, respectively. Each attack was executed 10 times and Kruiser detected all these overflows successfully.

In addition to the synthetic attacks, we also exploited two real-world heap buffer overflow vulnerabilities [28], [29] in Linux. For the first one, we sent particularly crafted ASN.1 BER data to trigger a heap overflow. In the second test, we used a special eCryptfs file whose encrypted key size is larger than ECRYPTFS_MAX_ENCRYPTED_KEY_BYTES to overflow a buffer. Kruiser detected all the realistic overflows.

The above experimental results indicate that Kruiser is effective in defending against kernel heap buffer overflow attacks.

8.2 Performance Overhead

To evaluate the performance of our monitoring mechanism, we carried out a set of experiments. First, we executed the micro-benchmark to measure the overhead at the kernel function call level. Then, we ran the SPEC CPU2006 Integer benchmark to test the application-level overhead. Each of these experiments was conducted in three different environments, including original Linux, Kruiser with SIM protection (referred as SIM-Kruiser subsequently), and Kruiser without SIM protection.

Micro-Benchmark. To evaluate the performance of the APIs exported by the slab allocator, we implemented a kernel module that invokes the APIs to allocate specific kernel objects with varied bytes (from 20 to 400 bytes). More specifically, we first used kmem_cache_create to create a specific cache. Then, we invoked kmem_cache_alloc twice to allocate two kernel objects from that cache. Since there was no available slab when the cache was just created, the first allocation triggered the construction of a slab, which included initialization of all the objects in this slab. For the second allocation, it just returned an already prepared object from the slab. Next, we released these objects by invoking kmem_cache_free. Finally, we issued kmem_cache_destroy to destruct the whole cache as well as its slab. To measure the execution time, we recorded the hardware time stamp counter register (with the instruction rdtsc) right before and after invoking these APIs. Table 1 shows the average execution time of Kruiser and SIM-Kruiser, which are normalized by the execution time of original Linux. From this table, we can see that there are three kernel APIs (i.e., kmem_cache_create, kmem_cache_alloc(2nd), and kmem_cache_free) with (or less than) 8% performance overhead. This is reasonable because we only add some small code to these functions. However, the kernel APIs kmem_cache_alloc(1st) and kmem_cache_destroy impose a little expensive overhead. There are two major sources for this overhead added by Kruiser and SIM-Kruiser: (1) generating secure canaries, and (2) checking canaries. Moreover, SIM-Kruiser incurs additional overhead caused by the context switch between the OS kernel and the In-VM monitor.

Application Benchmark. For application-level measurement, we executed the SPEC CPU2006 Integer benchmark suite. Fig. 10 shows that the average performance overhead for both Kruiser and SIM-Kruiser are negligible. When the slab allocation is frequent, the performance overhead is a little bit higher, such as in gcc; however, the maximal performance overhead is less than 3%.

TABLE 1
The average normalized execution time of kernel APIs with Kruiser and SIM-Kruiser when compared with original Linux.

Kernel APIs	Kruiser	SIM-Kruiser
kmem_cache_create	1.07	1.08
kmem_cache_alloc(1st)	5.13	5.76
kmem_cache_alloc(2nd)	1.05	1.06
kmem_cache_free	1.06	1.06
kmem_cache_destroy	4.37	4.95

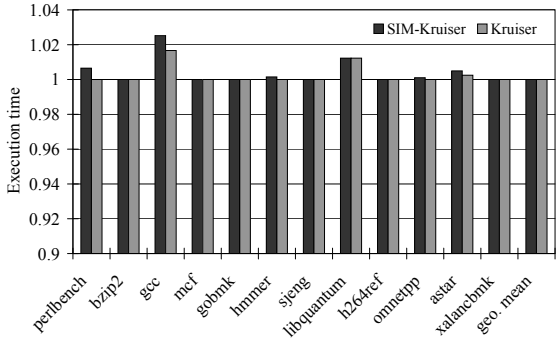

Fig. 10. SPEC CPU2006 performance (normalized to the execution time of original Linux).

Performance Cost to Dom0. To test the performance effect on the Dom0 VM, we measured the time to build the standard Linux kernel (linux-2.6.24) using the command `make`. The experiment shows the performance cost added by our monitor process is very small (i.e., less than 5%) when we compile the Linux kernel using a single thread. However, the performance overhead will be very obvious (i.e., more than 50%) when we compile the kernel using multiple threads. The main reason for this cost is that the compiling threads need to contend with the monitor process located in the Dom0 VM for the CPU resources. If the CPU resources are adequate, the add-on performance cost would be negligible. Otherwise, the Dom0 VM's performance would be affected to a certain degree.

Performance Cost on the 64-bit system. To facilitate monitoring the 64-bit system, we exploited the recent KVM hypervisor (version 2.9.0). To evaluate the performance cost on the 64-bit system, we also utilized the SPEC CPU2006 benchmarks. As shown in Fig. 11, the maximal cost of these benchmarks is less than 2%. Compared with the original 64-bit system, the performance cost introduced by our protection is very small.

8.3 Scalability

For the scalability measurement, we tested the throughput of the Apache web server with concurrent requests. Specifically, we ran Apache 2.2.8 to serve a 3.7KB html web page. We used ApacheBench 2.3 running on another machine—a Dell PowerEdge T310 Server with a 1.86G Intel E6305 CPU, 4 GB memory and Ubuntu 8.04 (linux-2.6.24)—to measure the Apache throughput over a GB LAN network. Each time we issued 10k http requests with various numbers of concurrent clients, and we observed that the number of the kernel heap buffer object allocation increases along with the concurrency level. As shown in Fig. 12, the performance overhead imposed by Kruiser and

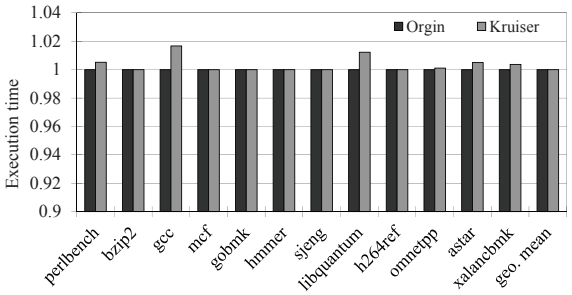

Fig. 11. SPEC CPU2006 performance on the 64-bit system (normalized to the execution time of original Linux).

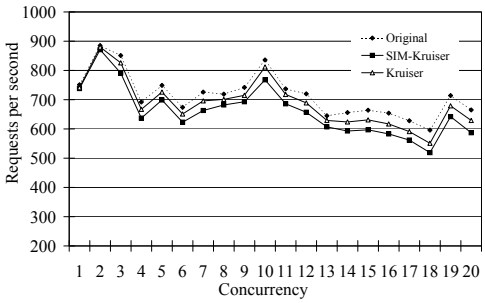

Fig. 12. Throughput of the Apache web server for varying numbers of concurrent requests.

SIM-Kruiser are both relatively stable. On average, Kruiser only incurs about 3.8% performance degradation and SIM-Kruiser about 7.9%.

In addition, we ran multiple VM instances that are monitored by one monitor process. Due to the user address space limitation, one monitor can only simultaneously check three VMs in maximum. Moreover, since the CPU resource is limited, each monitored VM was only allocated one VCPU. Before monitoring multiple VMs, the monitor needs to map the VM's kernel memory to its user address space. After that, we utilized PARSEC benchmarks to measure the performance overhead for each VM. Fig. 13 shows that the performance degradation is minimal when one monitor checks three VMs in parallel.

8.4 Memory Overhead

In addition to allocating the fixed-size memory for the PIA structure, Kruiser needs to add 4 bytes canary for each kernel heap object. To test the memory overhead introduced by our modified slab allocator, we developed a kernel module to invoke the kmalloc function to allocate a number of kernel objects, whose sizes are randomly selected between 1 to 1024 bytes. We recorded the number of memory pages used for the slab allocator right before and after these memory allocations. In this way, we can figure out the number of pages that are used for the newly allocated objects. Table 2 shows the added slab pages depending on the number of live allocated objects in the kernel heap. Compared with the native Linux, Kruiser introduce little memory overhead.

8.5 Detection Latency

We recorded the average cruising cycles (i.e., the average time for scanning all the PIA entries) for different applications in SPEC CPU2006, in order to evaluate the detection latency, which is less than or equal to the cruising cycle at the attack time. As

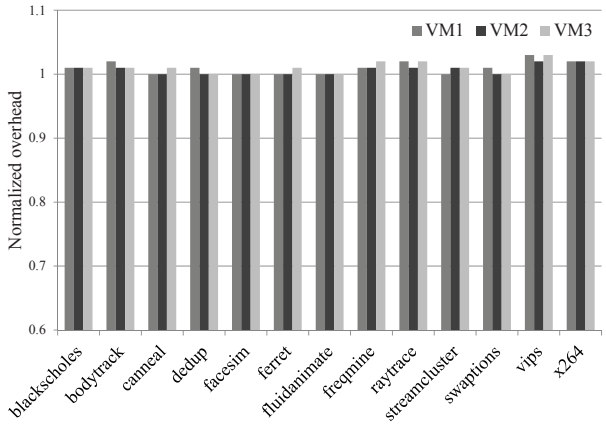

Fig. 13. PARSEC performance.

TABLE 2
Added slab pages for native Linux and Kruiser depending on the number of allocated kernel objects.

Kernel Heap Objects	Native Linux	Kruiser
1000	2	2
10,000	28	32
100,000	246	250

shown in Table 3, 10 of 12 applications' average cruising cycles are shorter than 29 ms, and the other two applications' are below 40 ms. We also recorded the number of scanned kernel objects in each cruising cycle. The results indicate that the average cruising cycle is mainly determined by the average number of scanned kernel objects. Let N be the number of scanned kernel objects and T the average time for the monitor process to check a kernel object. We have $C = NT$, where C is the cruising cycle. We can reduce the cruising cycle by keeping N small. One approach is to divide the PIA entries into different parts, and for each part, we create a separate monitor process. Another approach is to only monitor objects in general caches. This is practical because attackers mainly exploit this category of buffers in the real world.

During the cruising cycle, our monitor scans n kernel objects. We assume the scanning time for each object is the same, representing it as t. Moreover, the probability of scanning one object from n objects is assumed to be the same. Based on these assumptions, we can calculate the mathematical expectation E of the scanning time as follows:

$$E = \frac{1}{n}\sum_{i=1}^{n} i \times t = \frac{1+n}{2} \times t$$

In our experiment, the value of $n \times t$ is between 29ms-40ms, so the mathematical expectation is around between 14.5ms-20ms.

If an attacker can compromise the kernel via heap overflow within one cruising cycle, our monitor may not identify the heap corruption and then stop the attack instantly. However, thanks to the secure canary, the attacker may not be able to recover the corrupted canary within one cruising cycle. Thus, the heap attack will be detected after our monitor finishes scanning all the kernel objects.

9 Discussion and Limitations

Like other canary-based systems [6], [13], [75], our system cannot detect the attacks that only overwrite a critical location inside

TABLE 3
Different cruising cycle for different applications in the SPEC CPU2006 benchmark (The cruising number refers to the number of kernel objects that are scanned in each cruising cycle).

Benchmark	Maximum cruising number	Minimum cruising number	Average cruising number	Average cruising cycle(μs)
perlbench	107,824	105,145	106,378	39,259
bzip2	79,085	76,325	76,682	27,662
gcc	78,460	76,810	77,413	27,774
mcf	82,885	79,328	79,540	28,156
gobmk	80,761	80,345	80,519	28,606
hmmer	81,278	80,435	80,591	28,635
sjeng	81,437	80,259	80,535	28,610
libquantum	80,911	80,317	80,407	28,493
h264ref	80,756	80,337	80,480	28,572
omnetpp	82,109	80,796	81,088	28,836
astar	81,592	81,022	81,097	28,897
xalancbmk	99,436	82,747	88,454	30,190

the kernel object in concern without touching any canary. For example, a kernel module may invoke the kmalloc function to allocate a struct that contains a string buffer and a function pointer such that attackers could corrupt the function pointer by overflowing the neigbouring buffer. Moreover, if attackers are able to read the kernel memory by exploiting memory exposure vulnerabilities, they could also bypass our detection mechanism.

The second problem is that our system cannot detect the buffer overflow attacks that occur in the noncontiguous memory area. The main reason is our monitor only map and check the physically contiguous pages. To monitor noncontiguous memory buffers, we need to add hooks to the noncontiguous memory management. Specifically, the secure canary should be added to the end of the buffer. Then, the monitor should dynamically map the memory address for concurrent monitoring. Adding these functionalities to our system could increase some performance overhead. Fortunately, buffer overflows in the noncontiguous memory area is very rare in the real world.

In our current implementation, our system only supports the SLAB allocator for heap monitoring. To make our method work with the SLUB/SLOB allocators, some modifications to these allocators are needed. For example, we need to change the allocation metadata for recording the canary information. To add a canary into the end of a kernel buffer, we need to hook the corresponding SLUB/SLOB construction routines.

Although our prototype system is based on an earlier version of Linux (v2.6.24), our method could be applied to recent versions of Linux. For example, Linux recently introduces KASLR for randomizing the kernel code location in memory when the system boots. Since the SLAB allocator begins to work after the kernel code is loaded, our approach would be compatible with KASLR. To mitigate the Meltdown security vulnerability, kernel page-table isolation (KPTI) is introduced in recent Linux kernels. Our protection mechanism and KPTI can coexist in that KPTI does not need any modification to the SLAB allocator.

Due to the limited memory resource, we just carry out the experiment for a 64-bit system with 2 GB memory. For the experiment with a large memory (e.g., 64 GB), our method will introduce proportional memory overhead for storing the static PIA. Moreover, since most of the PIA entries are empty during the kernel heap monitoring, the detection latency will be increased. To address this issue, a potential method is to re-design the PIA structure. For example, the PIA structure could be re-designed by using three-level page tables, which contain hierarchical PIA directories and PIA entries. When the monitor process traverses along the PIA directory, it could bypass empty PIA entries corresponding to large bulks of contiguous pages.

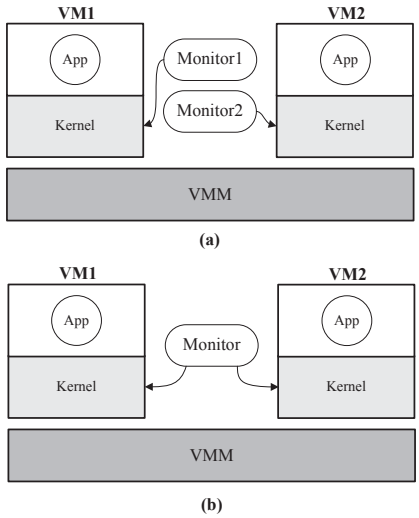

Fig. 14. Different schemes for system deployment.

Currently, our system cannot handle the case when the P2M table changes due to the page swap. To deal with this issue, we should hook the updating operation on the P2M table and then remap the machine frames for our monitor process. In addition, some modifications to the monitoring algorithm may be needed for the page mapping synchronization.

To deploy our system into cloud platforms, we have two different schemes, which is shown in Fig. 14. For the scheme a, we use one monitor to check different VMs. The main advantage of this scheme is to facilitate monitoring multiple VMs in parallel. However, this scheme will result in longer detection latency. For the scheme b, different VMs are monitored by different monitors. Compared with the first scheme, it may take more CPU resources, but the cruising cycle will be lower. To apply our method to the virtualization environment when the extended page table mechanism is enabled, we could take advantage of the recent in-VM monitoring framework [30] based on Intel EPTP-switching feature.

10 Related Work

10.1 Countermeasures Against Buffer Overflows

Over the past few decades, there has been extensive research in this area. We divided existing countermeasures against buffer overflows into seven categories: (1) buffer bounds checking [31], [32], [8], [9], [10], [12], [33], [34], [35], [36], (2) canary checking [6], [7], [13], (3) return address shadow stack or stack split [37], [38], [39], [40], [41], (4) non-executable memory [42], [43], (5) non-accessible memory [44], [45], [46], (6) randomization and obfuscation [47], [42], [48], [49], and (7) execution monitoring [50], [51], [52], [53], [15]. Few countermeasures are suitable for high performance kernel heap buffer overflow monitoring and no one has been deployed in production systems.

Kruiser falls into the category of canary checking. Canary was firstly proposed in StackGuard [6], which tackles stack-smashing attacks by putting a canary word before the return address on stack. A buffer overflow that overwrites the return address would

corrupt the canary value first. The approach has been integrated into GCC and Visual Studio. Robertson et al. [13] applied canary to protecting heap buffers. When a heap buffer is overrun, the canary of the adjacent chunk is corrupted, which, however, is not detected until the adjacent chunk is coalesced, allocated, or deallocated; i.e., the detection relies on the control flow. Larry H. [54] extended this idea to detect kernel heap buffer overflows. Different from these in-lined approaches, our approach enforces a constant concurrent canary checking and thus does not have the limitation. In addition, the *secure canary* conception is new.

The previous work Cruiser [18], among the existing countermeasures, first proposed concurrent buffer overflow cruising in user space using custom lock-free data structures. Unlike Cruiser that hooks per heap buffer allocation and deallocation, Kruiser explores the characteristics of kernel heap management to interpose the much less frequent operations that switch pages into and out of the heap page pool, such that our system relies on on a fix-sized array data structure instead of the lock-free data structures to maintain the metadata. The monitoring algorithms are thus very different. In addition, the hybrid monitoring scheme differs a lot from the user space monitoring.

Compared with the methods based on probabilistic memory safety (e.g., DieHard [55] and DieHarder [56]), Kruiser imposes negligible performance overhead. Nevertheless, Kruiser focuses on kernel heap, while DieHard and DieHarder have only been demonstrated for user-space programs. Our previous work Cruiser [18] on user-space buffer overflow monitoring presents detailed comparison with DieHarder on performance for the SPEC CPU2006 benchmark. In addition, DieHard and DieHarder consume more memory than Kruiser, which may be a problem for kernel.

10.2 Virtual Machine Introspection

Garfinkel and Rosenblum [57] first proposed the idea of performing intrusion detection from outside of the monitored system. Since then, out-of-VM introspection has been applied to control-flow integrity checking [58], [59], malware prevention, detection, and analysis [60], [61], [62], [63], [64], [65], [66], [67], [19], [68], [69], and attack replaying [70]. They monitor static memory areas (e.g. kernel code, Interrupt Description Table), interpose specific events such as page faults, trace system behaviors, or detect violations of invariants between data structures. Considering the volatile properties of heap buffers, these approaches are infeasible for kernel heap buffer overflow monitoring; for example, it is impractical to interpose every memory write on the heap. Some approaches detected buffer overflow attacks as a side effect by detecting corrupted pointers or control flows, but cannot deal with non-pointer and non-control data manipulation on heap buffer objects. Approaches, such as kernel memory mapping and analysis, can be misled by buffer overflow attacks or perform better without heap corruption. Our approach can be complementary to them providing lightweight heap buffer overflow detection.

In contrast to out-of-VM monitoring, SIM [17] puts the monitor back into the VM and enables secure in-VM monitoring by providing discriminative memory views for the monitored system and the monitor. Our approach makes use of this technique to protect the heap metadata, while the monitor process still runs out-of-VM to achieve parallel monitoring, leveraging the multiprocessor architecture. The hybrid scheme enables a secure and efficient monitoring.

10.3 OS Kernel Security

OSck [19] also performs kernel space cruising for rootkit detection. As OSck does not synchronize the running kernel and the verification process, it needs to suspend the system when an anomaly is detected to avoid false positives, while our approach does not need to stop the world for detection. In addition, OSck does not check generic buffers allocated using kmalloc, which are common attack targets, while Kruiser checks the whole kernel heap. Liu et al. [71] propose a hardware-based approach for concurrent introspection of guest VMs. This approach does not require any modification to the guest OS, and it combines the HTM hardware feature and virtualization to detect rootkits concurrently. Wang et al. [72] present a kernel rootkit detection technique that leverages the hardware performance counter and virtualization technique. Compared with these rootkit detection methods, our approach focuses on detecting kernel heap overflows.

kMVX [73] and UniSan [74] are developed to address the kernel information leak problems (e.g., out-of-bounds read), whereas our Kruiser design focuses on handling the out-of-bounds write problem. Similar to our work, KASAN [75] could be applied to detecting out-of-bounds write and use-after-free bugs in a Linux kernel. Since KASAN relies on inline security checks, it incurs considerable performance overhead. As a result, the KASAN tool may not be suitable for real-time kernel protection. To defend against data-oriented attacks in the kernel, Song et al. present a DFI protection system called KENALI [76] that is based on the LLVM compiler. Different from KENALI, our system utilizes the virtualization technology for kernel heap data monitoring, which requires minimal changes to the kernel.

11 Conclusion

We have presented KRUISER, a semi-synchronized concurrent kernel heap monitor that cruises over heap buffers to detect overflows in a non-blocking manner. Unlike traditional techniques that monitor volatile memory regions with security enforcement inlined into normal functionality (interposition) or by analyzing memory snapshots, we perform constant monitoring in parallel with the monitored VM on its live memory without incurring false positives. The hybrid VM monitoring scheme provides high efficiency without sacrificing the security guarantees. Attacks are bound to be detected within one cruising cycle. Our evaluation has shown that Kruiser imposes negligible performance overhead on the system running SPEC CPU2006 and 7.9% throughput reduction on Apache. The concurrent *kernel cruising* approach leverages increasingly popular multi-core architectures; its efficiency and scalability manifest that it could be deployed in cloud environment.

Acknowledgment

This work was partially supported by NSFC 61602035, AFOSR FA9550-07-1-0527 (MURI), ARO W911NF-09-1-0525 (MURI), NSF CNS-0905131, NSF CNS-1223710, and AFRL FA8750-08-C-0137.

References

[1] NIST. National Vulnerability Database, http://nvd.nist.gov/.
[2] sqrkkyu and twzi, “Attacking the core: Kernel exploiting notes,” 2007, http://phrack.org/issues.html.

[3] C. S. Technologies, "OpenBSD IPv6 mbuf remote kernel buffer overflow," 2007, http://www.securityfocus.com/archive/1/462728/30/0/threaded.

[4] T. Mandt, "Kernel pool exploitation on Windows 7," 2011, https://media.blackhat.com/bh-dc-11/Mandt/BlackHat_DC_2011_Mandt_kernelpool-wp.pdf.

[5] j00ru, "Exploiting a windows 10 pagedpool off-by-one overflow," 2018, https://j00ru.vexillium.org/2018/07/exploiting-a-windows-10-pagedpool-off-by-one/.

[6] C. Cowan and C. Pu, "StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks," in *Usenix Security '98*, January 1998, pp. 63–78.

[7] IBM, "ProPolice detector," http://www.trl.ibm.com/projects/security/ssp/.

[8] T. M. Austin, S. E. Breach, and G. S. Sohi, "Efficient detection of all pointer and array access errors," in *PLDI '04*, pp. 290–301.

[9] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of C," in *Usenix ATC '02*, June 2002, pp. 275–288.

[10] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "CCured: type-safe retrofitting of legacy software," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 3, pp. 477–526, 2005.

[11] R. W. M. Jones and P. H. J. Kelly, "Backwards-compatible bounds checking for arrays and pointers in C programs," in *the International Workshop on Automatic Debugging*, 1997.

[12] O. Ruwase and M. S. Lam, "A practical dynamic buffer overflow detector," in *NDSS '04*, pp. 159–169.

[13] W. Robertson, C. Kruegel, D. Mutz, and F. Valeur, "Run-time detection of heap-based overflows," in *LISA '03*, pp. 51–60.

[14] P. Argyroudis and D. Glynos, "Protecting the core: Kernel exploitation mitigations," in *Black Hat Europe '11*.

[15] B. Salamat, T. Jackson, A. Gal, and M. Franz, "Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space," in *EuroSys '09*, pp. 33–46.

[16] M. Dalton, H. Kannan, and C. Kozyrakis, "Real-world buffer overflow protection for userspace & kernelspace," in *Usenix Security '08*, pp. 395–410.

[17] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-VM monitoring using hardware virtualization," ser. CCS '09, pp. 477–487.

[18] Q. Zeng, D. Wu, and P. Liu, "Cruiser: Concurrent heap buffer overflow monitoring using lock-free data structures," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 367–377. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993541

[19] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel, "Ensuring operating system kernel integrity with OSck," ser. ASPLOS '11, pp. 279–290.

[20] Z. Wang and X. Jiang, "HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 380–395.

[21] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky, "HyperSentry: enabling stealthy in-context measurement of hypervisor integrity," in *Proceedings of the 17th ACM conference on Computer and communications security*, ser. CCS '10. New York, NY, USA: ACM, 2010, pp. 38–49.

[22] J. Wang, A. Stavrou, and A. Ghosh, "HyperCheck: a hardware-assisted integrity monitor," in *Proceedings of the 13th international conference on Recent advances in intrusion detection*, ser. RAID'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 158–177. [Online]. Available: http://dl.acm.org/citation.cfm?id=1894166.1894178

[23] W. Mauerer, *Professional Linux Kernel Architecture*. Wrox Press, 2008.

[24] D. Mosberger, "Memory consistency models," *Operating Systems Review*, vol. 17, no. 1, pp. 18–26, January 1993.

[25] P. E. Mckenney, "Memory barriers: a hardware view for software hackers," 2009.

[26] Wikipedia, "RC4," http://en.wikipedia.org/wiki/RC4.

[27] D. Roethlisberge, "Omnikey Cardman 4040 Linux driver buffer overflow," 2007, http://www.securiteam.com/unixfocus/5CP0D0AKUA.html.

[28] US-CERT/NIST, "CVE-2008-1673." [Online]. Available: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-1673

[29] ——, "CVE-2009-2407." [Online]. Available: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-2407

[30] B. Shi, L. Cui, B. Li, X. Liu, Z. Hao, and H. Shen, "Shadowmonitor: An effective in-vm monitoring framework with hardware-enforced isolation," in *Research in Attacks, Intrusions, and Defenses*. Cham: Springer International Publishing, 2018, pp. 670–690.

[31] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A first step towards automated detection of buffer overrun vulnerabilities," in *NDSS'00*, pp. 3–17.

[32] N. Dor, M. Rodeh, and M. Sagiv, "CSSV: towards a realistic tool for statically detecting all buffer overflows in C," in *PLDI '03*, June 2003, pp. 155–167.

[33] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors," in *Usenix Security '09*, pp. 51–66.

[34] E. D.Berger, "HeapShield: Library-based heap overflow protection for free," Univ. of Mass. Amherst, Tech. Report, 2006.

[35] T. K. Tsai and N. Singh, "Libsafe: Transparent system-wide protection against buffer overflow attacks," in *DSN '02*, pp. 541–541.

[36] K. Avijit and P. Gupta, "Tied, libsafeplus, tools for runtime buffer overflow protection," in *Usenix Security '04*, pp. 4–4.

[37] StackShield, 2000, http://www.angelfire.com/sk/stackshield/.

[38] T. Chiueh and F. Hsu, "RAD: A compile-time solution to buffer overflow attacks," in *ICDCS '01*, pp. 409–417.

[39] M. Prasad and T. Chiueh, "A binary rewriting defense against stack based buffer overflow attacks," in *Usenix ATC '03*, pp. 211–224.

[40] M. Frantzen and M. Shuey, "StackGhost: Hardware facilitated stack protection," in *Usenix Security '01*, pp. 55–66.

[41] J. Xu, Z. Kalbarczyk, S. Patel, and R. Iyer, "Architecture support for defending against buffer overflow attacks," in *Workshop Evaluating & Architecting Sys. Depend.*, 2002.

[42] The PaX project, http://pax.grsecurity.net/.

[43] Solar Designer, "Non-executable user stack," 1997, http://www.openwall.com/linux/.

[44] R. Hastings and B. Joyce, "Purify: Fast detection of memory leaks and access errors," in *the Winter 1992 Usenix Conference*, pp. 125–136.

[45] Valgrind, http://valgrind.org/.

[46] Electric Fence, "Malloc debugger," http://directory.fsf.org/project/ElectricFence/.

[47] E. Bhatkar, D. C. Duvarney, and R. Sekar, "Address obfuscation: an efficient approach to combat a broad range of memory error exploits," in *Usenix Security '03*, pp. 105–120.

[48] C. Cowan and S. Beattie, "PointGuard: protecting pointers from buffer overflow vulnerabilities," in *Usenix Security '03*, pp. 91–104.

[49] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi, "Randomized instruction set emulation to disrupt binary code injection attacks," in *CCS '03*, pp. 281–289.

[50] V. Kiriansky, D. Bruening, and S. P. Amarasinghe, "Secure execution via program shepherding," in *Usenix Security '02*, pp. 191–206.

[51] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *CCS '05*, pp. 340–353.

[52] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *OSDI '06*, pp. 147–160.

[53] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-variant systems: a secretless framework for security through diversity," in *Usenix Security '06*, pp. 105–120.

[54] L. H., "Linux kernel heap tampering detection," 2009, http://www.phrack.org/issues.html?issue=66&id=15.

[55] E. D. Berger and B. G. Zorn, "DieHard: probabilistic memory safety for unsafe languages," in *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '06. New York, NY, USA: ACM, 2006, pp. 158–168.

[56] G. Novark and E. D. Berger, "DieHarder: securing the heap," in *Proceedings of the 17th ACM conference on Computer and communications security*, ser. CCS '10. New York, NY, USA: ACM, 2010, pp. 573–584.

[57] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *NDSS '03*, pp. 191–206.

[58] N. L. Petroni, Jr. and M. Hicks, "Automated detection of persistent kernel control-flow attacks," ser. CCS '07, pp. 103–115.

[59] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," ser. SOSP '07, pp. 335–350.

[60] K. Kourai and S. Chiba, "HyperSpector: virtual distributed monitoring environments for secure intrusion detection," ser. VEE '05, pp. 197–207.

[61] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Antfarm: tracking processes in a virtual machine environment," ser. Usenix ATC '06.

[62] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: malware analysis via hardware virtualization extensions," ser. CCS '08, pp. 51–62.

[63] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," ser. Oakland '08, pp. 233–247.

[64] A. Lanzi, M. I. Sharif, and W. Lee, "K-Tracer: A system for extracting kernel malware behavior," in *NDSS '09*.

[65] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang, "Mapping kernel objects to enable systematic integrity checking," ser. CCS '09, pp. 555–565.
[66] J. Rhee, R. Riley, D. Xu, and X. Jiang, "Kernel malware analysis with untampered and temporal views of dynamic kernel memory," ser. RAID'10, pp. 178–197.
[67] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang, "SigGraph: Brute force scanning of kernel data structure instances using graph-based signatures," ser. NDSS '11.
[68] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," ser. Oakland '11.
[69] P. Mishra, V. Varadharajan, E. Pilli, and U. Tupakula, "Vmguard: A vmi-based security architecture for intrusion detection in cloud environment," *IEEE Transactions on Cloud Computing*, 2018.
[70] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen, "Detecting past and present intrusions through vulnerability-specific predicates," ser. SOSP '05, pp. 91–104.
[71] Y. Liu, Y. Xia, H. Guan, B. Zang, and H. Chen, "Concurrent and consistent virtual machine introspection with hardware transactional memory," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, Feb 2014, pp. 416–427.
[72] X. Wang and R. Karri, "Reusing hardware performance counters to detect and identify kernel control-flow modifying rootkits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 3, pp. 485–498, March 2016.
[73] S. Österlund, K. Koning, P. Olivier, A. Barbalace, H. Bos, and C. Giuffrida, "kmvx: Detecting kernel information leaks with multi-variant execution," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. ACM, 2019, pp. 559–572.
[74] K. Lu, C. Song, T. Kim, and W. Lee, "Unisan: Proactive kernel memory initialization to eliminate data leakages," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. ACM, 2016, pp. 920–932.
[75] T. L. Kernel, "The kernel address sanitizer (kasan)." [Online]. Available: https://www.kernel.org/doc/Documentation/dev-tools/kasan.rst
[76] C. Song, B. Lee, K. Lu, W. R. Harris, T. Kim, and W. Lee, "Enforcing Kernel Security Invariants with Data Flow Integrity," in *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

Donghai Tian is an Assistant Professor in School of Computer Science and Technology, Beijing Institute of Technology, China. His research interest lies in security issues in computer systems and networks, including areas ranging from operating system security, software security and network security, to virtualization technologies. From 2009 to 2011, he was a visiting student in Pennsylvania State University, USA. He received his Ph.D. degree from Beijing Institute of Technology, China in 2012.

Qiang Zeng is an Assistant Professor in the Department of Computer Science and Engineering at University of South Carolina. He received his Ph.D. degree from Pennsylvania State University. He currently works on cloud, smartphone security and self-healing software. He is also interested in emerging security challenges in other areas. He has rich industry experience and has worked in the IBM T.J. Watson Research Center, the NEC Lab America, Symantec and Yahoo.

Dinghao Wu is an Associate Professor in College of Information Sciences and Technology, Pennsylvania State University. His research interest includes software security and programming languages. Prior to joining Penn State, he was a research engineer at Microsoft in the Center for Software Excellence and the Windows Azure Division. He received his Ph.D. degree from Princeton University in 2005.

Peng Liu received his BS and MS degrees from the University of Science and Technology of China, and his PhD from George Mason University in 1999. He is a Professor of Information Sciences and Technology, founding director of the Center for Cyber Security, Information Privacy, and Trust, and founding director of the Cyber Security Lab at Penn State University. His research interests are in all areas of computer and network security. He has published a monograph and over 220 refereed technical papers. His research has been sponsored by NSF, ARO, AFOSR, DARPA, DHS, DOE, AFRL, NSA, TTC, CISCO, and HP. He has served on over 90 program committees and reviewed papers for numerous journals. He is a recipient of the DOE Early Career Principle Investigator Award. He has co-led the effort to make Penn State a NSA-certified National Center of Excellence in Information Assurance Education and Research. He has advised or co-advised around 20 PhD dissertations to completion.

Changzhen Hu received his Ph.D. degree from Beijing Institute of Technology, China in 1996. He holds the Professor and the Associate Dean of School of Computer Science and Technology, Beijing Institute of Technology, China. His research interest includes information security, network, and pattern recognition.