Software Cruising: A New Technology for Building Concurrent Software Monitor

Dinghao Wu, Peng Liu, Qiang Zeng, and Donghai Tian

Abstract We introduce a novel concurrent software monitoring technology, called *software cruising*. It leverages multicore architectures and utilizes *lock-free* data structures and algorithms to achieve efficient and scalable security monitoring. Applications include, but are not limited to, heap buffer integrity checking, kernel memory cruising, data structure and object invariant checking, rootkit detection, and information provenance and flow checking. In the software cruising framework, one or more dedicated threads, called cruising threads, are running concurrently with the monitored user or kernel code, to constantly check, or cruise, for security violations. We believe the software cruising technology would result in a game-changing capability in security monitoring for the cloud-based and traditional computing and network systems.

We have developed two prototypical cruising systems: *Cruiser*, a lock-free concurrent heap buffer overflow monitor in user space, and *Kruiser*, a semi-synchronized non-blocking OS kernel cruiser. Our experimental results showed that software cruising can be deployed in practice with modest overhead. In user space, heap buffer overflow cruising incurs only 5 % performance overhead on average for the SPEC CPU2006 benchmark, and the Apache throughput slowdown is only 3 % maximum and negligible on average. In kernel space, it is negligible for SPEC, and 3.8 % for Apache. Both technologies can be deployed in large scale for cloud data centers and server farms in an automated manner.

D. Wu (✉) • P. Liu • Q. Zeng
Pennsylvania State University, University Park, PA 16802, USA
e-mail: dwu@ist.psu.edu; pliu@ist.psu.edu; quz105@psu.edu

D. Tian
Beijing Institute of Technology, Beijing, China
e-mail: dhai@bit.edu.cn

S. Jajodia et al. (eds.), *Secure Cloud Computing*, DOI 10.1007/978-1-4614-9278-8_14,

1 Introduction

Existing security-related software monitoring techniques could be roughly broken down into two categories: control-receiving monitoring and non-control-receiving monitoring. Control-receiving monitoring is well captured by the classic concept of reference monitors. A reference monitor defines a set of requirements that governs the reference validation mechanism. As stated by Schneider [50], "A reference monitor is guaranteed to receive control whenever any operation in some specified set is invoked." This category can be further classified into several classes. For examples,

- Operating system kernels as a reference monitor for operations on system objects (e.g., files and processed).
- Memory mapping hardware as a reference monitor (for accesses to memory pages).
- Processors as a reference monitor. Using tagging memory support, enforcement of information flow security policies could be pushed into the processor itself [81].
- Inlined reference monitors such as Software-based Fault Isolation (SFI) [75] and Jif [35]. Through static instrumentation, SFI can monitor a distrusted module writing or jumping to an address outside its fault domain. Enforced at both compile time and run time, Jif can impose information flow control and access control.
- Dynamic taint analysis (DTA) as a reference monitor [37]. Through static or dynamic instrumentation, or a combination of static and dynamic instrumentation, DTA can monitor data flows among instructions at byte-level granularity.

Non-control-receiving monitoring is not always bounded with control receiving. Due to various reasons (e.g., performance overhead), quite a few classes of monitoring do not expect to receive any control. Their primary goal is to obtain some specified awareness of the system being protected. Control-receiving monitoring is active monitoring; in contrast, non-control-receiving monitoring is passive monitoring. For examples,

- OS level monitors can collect system call traces for intrusion detection [23] and backtracking purposes [28].
- Calling context monitors can obtain the calling context information of an application for performance analysis and debugging purposes.
- Memory performance (e.g., memory leak) monitors can obtain awareness about certain memory leak problems.
- Architecture level monitors (e.g., shadow gates [67]) could be added to track information flows.

Fine-grained software monitoring or security enforcement, such as inlined reference monitor, is often inlined, which delays the execution of the protected programs. In addition, inlined monitor code runs in the same address space as the

program being monitored. This could cause safety and security issues. The inlined code may introduce security holes or cause robustness problems. If the monitor code fails, the original program will fail as well. If the monitor code is blocked, the original program is often blocked as well. Furthermore, it is difficult to monitor or enforce concurrency properties with inlined code since the monitor code is scattered and often needs additional synchronization.

It is quite challenging to parallelize control-receiving monitoring. An example is dynamic taint analysis, for which parallel monitoring is still not very practical primarily due to the pervasive data and control dependence among the monitor and normal program execution. This has been a main cause of high performance overhead, a major obstacle to adopt concurrent software monitoring in practice.

The performance overhead of concurrent monitors comes from two sources: logging/monitoring and synchronization between the monitored code (logging) and monitor threads (monitoring). The latter has implicit blocking cost if the synchronization primitives used are lock-based. When the monitor threads are blocked due to external events, such as IO and OS preemptive scheduling, the threads being monitored will also be blocked in a lock-based synchronization style even if the monitor threads are not monitoring.

Our key insight is that we can explore multicore architectures for concurrent security monitoring using novel lock-free (non-blocking) data structures and algorithms[1] to eliminate blocking cost and thus make the concurrent monitoring extremely attractive in terms of performance and scalability. Since the synchronization between the original program and the monitor is non-blocking, this also makes the monitoring system monitor kill-safe; that is, the original program won't fail even if the monitor is blocked or crashed.

2 Software Cruising

Software Cruising is a novel concurrent software monitoring technology that migrates security enforcement from the monitored code, either in user or kernel space, to a concurrent monitor thread. The technology leverages multicore and multiprocessor architectures and uses *lock-free* data structures and algorithms to achieve *non-blocking* and efficient synchronization between the monitor and monitored code.

[1]Technically speaking, lock-free and non-blocking are related, but different concepts. Here, we do not distinguish the difference and rather use them interchangeably to mean that it is not traditional lock-based and not blocking.

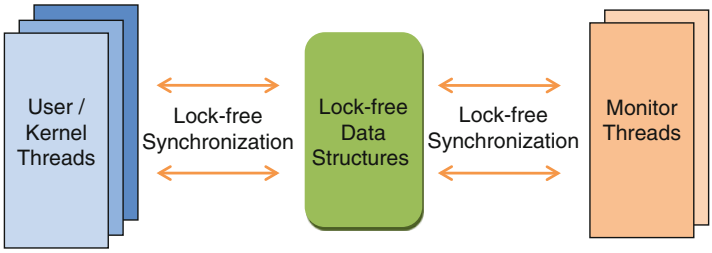

Fig. 1 The software cruising architecture

2.1 Architecture

In the software cruising framework, one or more dedicated threads, called *cruising threads*, are running concurrently with the monitored user- or kernel- code to constantly check, or *cruise*, for security violations. Figure 1 shows the architecture. It leverages the increasingly popular multicore architectures and lock-free (non-blocking) synchronizations.

The lock-free data structure is used to log information necessary for security monitoring. The monitored code in either user or kernel space and the monitor threads do not communicate directly, but rather through the lock-free data structures using non-blocking synchronization primitives. The key is to use lock-free data structures and algorithms to achieve the non-blocking property between the monitor and the code being monitored. The monitor thread(s) are always checking (cruising), possibly in spare cores on multicore processors, for security violations, but the user/kernel threads' executions are not blocked.

2.2 Features

The proposed software cruising technology has a number of distinct features that make it very attractive.

Leveraging Multicore Architectures for Concurrent Security Monitoring

The software cruising technology leverages multicore architectures with *lock-free* and *non-blocking* synchronization for security monitoring and enforcement. As the monitor threads running on separate cores, the execution of the original program is not largely affected, with loose coupled lock-free synchronization. With the increasingly popular multicore and multiprocessor architectures, this can minimize the performance overhead on the program being monitored. This also makes deployment in the cloud environment easier and more flexible since the monitor code can be run in separate virtual machines.

Protecting Monitor Threads from Malware

Malware, e.g., compromised user/kernel threads or untrusted kernel extensions hosting rootkits, could poison a monitor thread. To protect monitor threads, we could apply a new technology we developed recently [78]. Via HAP (hardware-assisted paging), this technology forces different subjects (e.g., user/kernel threads, monitor threads, untrusted kernel extensions, trusted kernel extensions, kernel core) to use different sets of page tables. The set of pages used by the monitor threads and the lock-free data structures can be flagged as unreadable, unwritable, or unexecutable as needed so that it can be protected from other threads running in the protected mode. Moreover, to prevent the monitor from being tampered and provide guaranteed performance isolation, we can utilize the virtualization technology and apply the SIM framework [58] to run the monitor process out of the monitored VM, while collecting heap memory allocation information inside the monitored VM in a secure and efficient way.

Non-blocking and Lock-Free

Our design is completely non-blocking between the monitor and the monitored program. Even if the monitor is blocked due to external IO events or OS preemptive scheduling, the program execution can still make progress without waiting. One of our designs for user-space heap buffer overflow monitoring adopts lock-free data structures. Our design of kernel cruising is semi-synchronized, but ensures correctness and non-blocking. Advantages of the lock-free and semi-synchronized designs include efficiency, scalability, deadlock-free, and kill-safe (see next paragraph on kill-safe).

Monitor Kill-Safe

Since our design is non-blocking and lock-free, it is safe to kill the monitor and any other cruising threads. We call this feature monitor kill-safe. In large-scale distributed systems such as cloud computing, hardware and software could fail frequently. The monitor kill-safe feature is particularly attractive in such scenarios.

Efficiency

The program being monitored incur very low performance overhead because (1) the monitoring code is in separate threads (possibly) running on separate cores, and (2) all communications are non-blocking so that even if the monitor is blocked due to external IO events or OS preemptive scheduling the program execution can still make progress without waiting.

Scalability

As software becomes more and more concurrent with more cores from hardware, the synchronization cost is more likely to become a bottleneck. The software cruising framework scales much better in this scenario since cruisers and user programs are running concurrently in a lock-free non-blocking manner.

One-to-One and One-to-Many Virtual Machine (VM) Monitoring

Software cruising can be deployed in large-scale (cloud) data centers and server farms. The software cruising framework has very flexible deployment options such as one-to-one and one-to-many monitoring. The one-to-one scheme is that one monitor corresponds to one virtual machine cruising, while the one-to-many is that one monitor cruises for multiple virtual machines. The one-to-many scheme is especially attractive for monitoring large-scale clouds.

2.3 Applications

With these distinct features, we sketch out a number of applications of software cruising. Software cruising can be applied to both user-space and kernel-space software monitoring. We have conducted two cases studies, one in user space and the other in kernel space. The security property we choose to monitor is heap buffer overflow. In user space, we developed *Cruiser*, a lock-free concurrent heap buffer overflow monitor (See Sect. 3 for more details). In kernel space, we developed *Kruiser*, a semi-synchronized non-blocking OS kernel cruiser (See Sect. 4 for more details).

Software cruising has flexible deployment options. It can be applied to application and system software running in a single computer, as well as large-scale distributed and networked systems such as data centers in the cloud computing environment. In such scenario, the monitor can be run in separate virtual machines with different protection level and makes the cruising system more scalable and more secure.

Other applications of software cruising include, but are not limited to, data structure and object invariant checking, rootkit detection, and information provenance and flow checking. For some good engineering reasons, low-level system code often contains features, e.g. custom linked list, that are hard to abstract and verify statically [11, 13, 31]. Instead, we can apply software cruising to dynamically check invariants; that is, we cruise to check that the data structure in memory is a well-formed.

3 Cruiser: Lock-Free Concurrent Heap Buffer Overflow Monitoring

In this section, we introduce the design of *Cruiser*, a lock-free concurrent heap buffer overflow monitor in user space. Interested readers are referred to Zeng, Wu, and Liu [82] for more technical details.

3.1 Introduction

Buffer overflow attacks are often the first step taken by multistage exploits. For example, the multistage attack example shown in MulVAL [40] starts with either CVE-2002-0392 [71] or CVE-2003-0252 [72], both are buffer overflow related vulnerabilities. Despite many counter measures developed, buffer overflow based attacks are still a great threat.

As a case study, we have applied software cruising to the heap buffer overflow problem and developed a novel dynamic heap buffer overflow detector, called *Cruiser*. The key ideas are (1) to create a dedicated monitor thread, which runs concurrently with user threads to cruise over, or keep checking constantly, dynamically allocated buffers against overflows; and (2) to utilize lock-free data structures and non-blocking algorithms, through which user threads communicate with the monitor thread with minimum overhead and without being blocked. The first idea leverages increasingly popular multicore architectures for security monitoring, and the second minimizes the communication and synchronization cost by removing the blocking overhead.

3.2 Design

Our method is canary-based [15]. Each dynamically allocated buffer is surrounded by two canary words; as long as a canary is found corrupted, an overflow is detected. Buffer addresses are collected in a lock-free data structure efficiently without blocking user threads. By traversing the data structure, buffers on heap are under constant surveillance of the concurrent monitor thread.

Cruiser Architecture

To efficiently maintain dynamic memory allocation information, we design the cruiser architecture such that the communication between the original program and the monitor is loosely coupled and non-blocking. As shown in Fig. 2, malloc calls are intercepted to allocate additional space for canary and place the allocated buffer information onto a list of ring data structures. There is one ring per user thread so

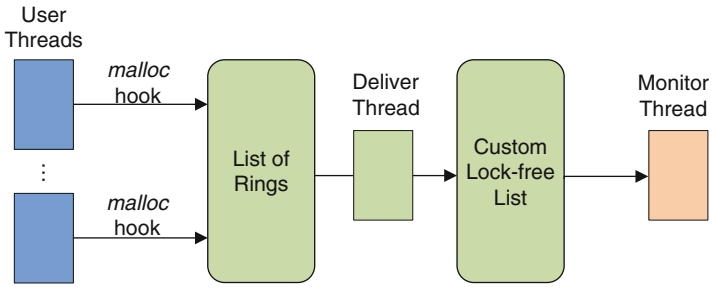

Fig. 2 The cruiser architecture

that there is no race conditions between two malloc calls. The malloc calls then return promptly, and one or several deliver threads move the metadata from rings to a custom lock-free linked list. The monitor thread cruises over the segmented list to check buffer overflows.

Ring

The ring data structure is based on the single-producer single-consumer FIFO wait-free ring buffer proposed by Lamport [32]. This algorithm allows a producer and a consumer to operate concurrently, with very low synchronization overhead as the producer and the consumer are synchronized via simple read/write instructions on the two control variables, the ring head and tail.

Segmented Lock-Free Linked List

Our custom lock-free linked list is segmented. The list consists of segments, each of which is a linked list itself. We construct one segment for each user thread to minimize the race conditions on list operations. Each segment has a dummy node head which is never removed. Also, the first non-dummy node will not be deleted until a new node is inserted before it. Thus the lock-free node insertion after the dummy node can be simply implemented using an atomic compare-and-swap (CAS) instruction. The buffer release and node deletion in the lock-free linked list is more complicated and we refer readers to our Cruiser paper [82] on the technical details. This custom lock-free linked list is very efficient and has the following distinct features: (1) Wait-free access and zero-contention; (2) No ABA problem [25]; and (3) No need to use special memory reclamation such as reference counters or hazard pointers [33].

3.3 Results

We evaluated Cruiser on its effectiveness, execution overhead, and scalability with varying number of threads.

Effectiveness

We tested the effectiveness of Cruiser on the SAMATE Reference Dataset (SRD) [38], as well as a set of well-known real-world exploits (wu-ftpd [51], Sudo [52], CVS [53], libHX [55], Lynx [56], and Firefox [54]). The experiments show that Cruiser can detect all the overflows, duplicate and invalid buffer frees.

Performance Overhead

We evaluated the performance overhead of Cruiser with the SPEC CPU2006 Integer benchmark suite. The results show that Cruiser incurs very low execution overhead: 5 % on average for the eager buffer release option and 12.5 % for the lazy option.

Scalability

We also evaluated Cruiser on the multithreaded setting. We configured the Apache web server with different number of concurrent requests (from 1 to 110) and tested Cruiser's scalability. The experimental results show that Cruiser scales well. The maximum slowdown of the Apache throughput is about 3 % and the average slowdown is negligible.

4 Kruiser: Semi-synchronized Non-blocking OS Kernel Cruising

In this section, we introduce the design of *Kruiser*, a semi-synchronized non-blocking OS kernel cruiser. Interested readers are referred to Tian et al. [66] for more technical details.

4.1 Introduction

It is desirable to adopt software cruising to monitor OS kernel memory integrity and other safety and liveness properties. The lock-free and non-blocking properties

of software cruising are especially attractive in kernel space since there are many tasks, events, and execution threads working simultaneously in kernel. If we use lock-based synchronizations for monitoring, it is likely that it will affect the kernel performance and execution characteristics.

Cruiser, as presented in the previous section, cannot be directly applied to monitor kernel buffer overflows due to the following reasons: (1) user- and kernel-space heap management schemes are quite different; (2) the runtime execution characteristics of kernel is quite different from user programs; and (3) OS kernel usually is not just one standalone program like typical user-space programs.

We have developed a prototype—called *Kruiser*, which stands for *kernel cruising*—that can monitor integrity of OS kernel memory. In kernel space, objects (or buffers) with the same size (from kernel or user-space programs) are usually allocated in the same page(s). Kruiser leverages this kernel memory management characteristic information and cruises over pages at first level, and individual buffers at the second level. Kruiser poses minimal changes to the existing OS kernel and can be deployed in large-scale cloud data centers to monitor many virtual machines scalably with the one-to-many virtual machine monitoring scheme.

4.2 Design

Kernel space presents new and more difficult challenges in designing software cruising systems.

Challenges

Synchronization

Synchronization is vital to ensure the monitor process locate and check live buffers efficiently and reliably without incurring false positives. To achieve highly efficient concurrent monitoring, we explore page-level information and design a semi-synchronized algorithm which introduces zero contention into kernel operations and performs non-blocking heap monitoring without incurring false positives or suspending the system.

Self-Protection

As a countermeasure against buffer overflow attacks, our component can become an attack target itself. We rely on a monitor process that keeps checking constantly—that is, cruising—the kernel heap integrity. This busy process can be an explicit attack target. By killing the monitor process, attackers completely disable the detection. Attackers can also tamper the data structure needed by our

component to mislead or evade the detection. Thus we need to protect the safety of the monitor process and ensure the integrity of related data structures. To address this challenge, we apply the virtualization technology to deploy the monitor process

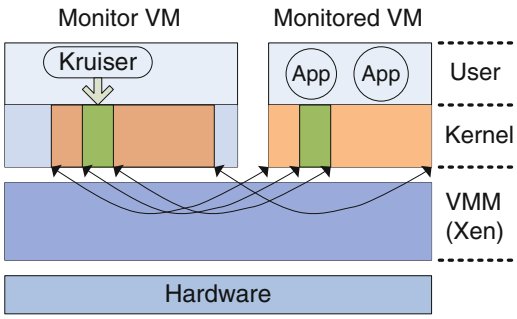

Fig. 3 The Kruiser architecture (using virtualization and direct memory mapping)

into a trusted environment. To ensure the same efficiency as in-the-box monitoring, we leverage the Direct Memory Mapping (DMM) technique, which allows the monitor process to access the monitored OS memory. To protect our data structure from being overflowed or underflowed, we apply two write-protected pages surrounding the data structure.

Architecture

Kruiser attaches one canary word at the end of each heap buffer and runs a separate monitor process, which keeps scanning, or cruises, the canaries to detect buffer overflows and runs concurrently with the monitored system. As shown in Fig. 3, Kruiser, or the monitor process, is run in a separate VM than the monitored OS to strengthen self-protection. The heap buffer metadata is kept in the monitored VM to achieve efficient updating. The monitor cruises over the heap metadata via an efficient technique called direct memory mapping. Once a kernel heap buffer canary is found corrupted, an overflow is reported.

The design of Kruiser is based on Linux and the Xen hypervisor. The Kruiser system can be divided into three parts: VMM, Dom0 VM, and DomU VM (the monitored VM). Dom0 VM contains the monitor process and the custom driver, which reside in user space and kernel space, respectively. The custom driver is used to help the monitor process release memory *but with its page tables retained*. A tiny component, namely Memory Mapper, inside the VMM is used to map the kernel memory of the monitored VM to the page table entries retained by the custom driver. A static array, called Page Identity Array (PIA), stores all the metadata at page level, and the interposition code reside in the kernel space of DomU VM.

Kernel Cruising

Kruiser keeps the metadata at page level and stores them in a static array called Page Identity Array (PIA). This array, however, can incur a variety of race conditions and atomicity issues. Introducing additional complex synchronization on PIA will inevitably affect the kernel performance. Instead, we design a novel algorithm that leverages kernel behavior to resolve the race conditions. To avoid race conditions on concurrent PIA entry updates, we leverage the critical section that are already exist in the kernel code that adds or removes a page from the page table to get a free ride with negligible cost for the PIA array entry update. Concurrent PIA entry read and write may cause inconsistent values being used. Instead of avoiding this read-write race condition, we let it occur, but avoid using inconsistent values by detecting inconsistent version numbers. Each PIA entry contains a version number which is incremented whenever the page corresponding to the PIA entry is added or removed from the heap page pool. The inconsistent values can be detected by comparing the version numbers before and after the read.

This non-blocking algorithm is constructed using simple reads, writes, and memory barriers without complicated and expensive synchronization mechanisms. The monitor process is lightly synchronized by reading version numbers twice, while other processes manipulating heap pages make progress without being synchronized or blocked by the monitor. In other words, the synchronization is one-way. That is why we call it *semi-synchronized non-blocking kernel cruising*. It is semi-synchronized in another sense. On the 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 to freeze the entire system for recheck, but still *does not incur any false positives*.

4.3 Results

To evaluate Kruiser, we developed a prototype based on 32-bit Linux and the Xen hypervisor.

Effectiveness

We conducted effectiveness tests on three vulnerabilities [47, 62] deliberately introduced in the Linux kernel and two real-world heap buffer overflow vulnerabilities [69, 70] in Linux. Our experimental results indicate that Kruiser is effective in defending against kernel heap buffer overflow attacks.

Performance Overhead

We evaluated the performance overhead of Kruiser on the SPEC CPU2006 benchmark. Our results showed that the average execution performance overhead is

negligible. When the slab allocation is frequent, the performance overhead is a little bit higher, such as in gcc, but the maximal performance overhead is still less than 3 %.

Scalability

We also evaluated the scalability of Kruiser on the Apache server in a multithreaded setting. The setup is similar to that of Cruiser (see Sect. 3.3). Our experimental results showed that the average slowdown of the Apache throughput is 3.8 and 7.9 % for a more secure Kruiser option.

5 Discussion

In this section, we discuss several advanced options and potential future application of software cruising.

5.1 Detection Latency

Since our software cruising is non-blocking, our monitoring does not suspend the system being monitored for detection. Thus, the detection latency becomes a critical indicator of the detection effectiveness. The time it takes a software cruising system to complete whole system monitoring once is called cruising cycle. It is important to keep the cruising cycle short so that we can detect an exploit quick enough. For the two applications we developed, the cruising cycles are both tunable; that is, we can configure the software cruising systems make small cruising cycle. Cruiser can achieve this with more than one monitor thread and keep the cruising list short enough. Each monitor thread only cruises part of the linked list. This can be achieved easily since the lock-free linked list is segmented. Kruiser can achieve this in a similar way. We can logically divide the metadata data structure into several segments and deploy equal number of monitor threads, so that each monitor thread only needs to be responsible for one segment.

5.2 Guaranteed Detection

Our cruising systems race with attackers: As long as an exploit cannot succeed within a cruise cycle after a canary is corrupted, it is bound to be *prevented*. In addition, even an attacker has compromised the system by exploiting a (kernel) heap buffer overflow vulnerability and enabled a remote shell with root privileges, the canary corrupt should be detected before the attacker keys in the first command, since a cruise cycle is normally less than a few milliseconds. In this sense, we “raised

a bar" for attackers. However, automatic attack vectors such as worms can be fast and advanced attacks may directly manipulate our data structures or try to recover the corrupted canaries using the keys. Moving the data structures and keys to a separate VM gains security but can lead to high performance overhead. Instead, we combine Secure-In-VM (SIM) [58] and *secure canary generation* to prevent attackers from recovering the corrupted canary, even after the system has been compromised and entirely controlled by attackers.

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 metadata; and (2) The canary generation based on the stream cipher guarantees the difficulty for attackers to recover the corrupted canaries within one cruising cycle. 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 and which is very rare.

Here we assume that the attacker does not reboot the system after a successful exploit to evade detection. We can add an additional cruiser check, to see whether the system has been compromised or not, in the system shutdown (reboot) routine to relax this assumption. Combined with the checkpoints technique, this guarantee enables a system to recover the nearest clean state.

5.3 Cloud Cruising

The software cruising approach leverages increasingly popular multicore architectures; its efficiency and scalability show that it can be applied to data centers and server farms in practice. Cruiser can be applied to shrink-wrapped software in an automated manner with negligible cost. The scheme in our prototype Kruiser is one-to-one monitoring on VMs. An advanced option in this design space is the one-to-many scheme; that is, one VM (monitor) cruises over multiple VMs, especially for the VMs that reside in the same physical machine. This is vital to the scalable online monitoring for cloud data centers and server farms.

Large data centers using shipping-containers packed with thousands of servers each are common nowadays. Therefore, scalable deployment is a critical requirement for intrusion detection measures in data centers. Unlike traditional interposition-based monitors, which may intervene normal functionalities frequently, Kruiser imposes minimal interference and performs monitoring in parallel with the monitored VM. Moreover, with the one-to-many option, one Kruiser instance is able to monitor multiple VMs given an acceptable detection latency much longer than the cruising cycle, without affecting the guaranteed detection property. In addition, the performance isolation provided by the underlying VMM ensures the monitor process and the monitored VM do not abuse computing resources to interfere with each other, which is a desirable property for users.

With the popularity of multicore architectures, servers built with many cores are more and more common. The hardware evolution trend embraces the concurrent monitoring fashion, as the cost for a unit core running a monitor instance decreases sharply, and the extra energy consumption by one core is relatively low for machines with hundreds of cores. Therefore, the scalability and low cost properties imply that Cruiser and Kruiser can be practically applied to large data centers and server farms, as one of the intrusion detection instruments in practice.

6 Related Work

In this section, we present related work on buffer overflow detection, system integrity, information flow integrity, and self-healing software.

6.1 Buffer Overflow Detection

Over the past few decades, there has been extensive research in this area, including buffer bounds checking [2–4, 7, 18, 26, 36, 48, 68, 74], canary checking [15, 24, 46], return address shadow stack or stack split [12, 21, 44, 64, 79], non-executable memory [61, 65], non-accessible memory [20, 22, 73], randomization and obfuscation [6, 9, 14, 65], and execution monitoring [1, 10, 16, 29, 49].

Despite so many countermeasures, only a few of them, such as StackGuard [15], ASLR (Address Space Layout Randomization) [9, 65], NX memory [61, 65], and DieHard [8] and DieHarder [39], are widely deployed in production systems. In Table 1, we compare Cruiser with those widely deployed tools and techniques.

Table 1 Comparison of some widely deployed tools and technologies with Cruiser

	Stack-Guard	ASLR	NX	DieHard & DieHarder	Cruiser
Low performance overhead	●	●	●	○	●
Easy to deploy and apply	●	●	●	○	●
No false alarms	●	●	●	●	●
Mainstream platform compatible	●	●	●	●	●
Program semantics loyalty	●	●	●	●	●
Legacy code compatible	●		●		●
Binary code compatible			●		●
No need for recompilation			●		●
Able to locate corrupted buffers				●	●
Leveraging multicore architectures					●
Guaranteed detection or prevention		○	●	○	●
Deployed to the field	●	●	●	●	⋆

Legend: ● means positive; ○ means partially or almost; ⋆ means just open-sourced

Software cruising shares many features with these techniques, including low performance overhead, easiness to deploy and apply, no false alarms, compatibility with mainstream platforms, and program semantics loyalty.

Cruiser bears many similar features with StackGuard [15]. Kruiser exhibits excellent performance on system kernel integrity checking, with novel features such as *secure monitor protection* and *guaranteed detection* (even after an initial successful exploit). In addition, Software cruising systems also have the following features: non-blocking and lock-free monitoring, monitor kill-safe, compatibility with legacy code, no need for recompilation, i.e. working with binary executables, ability to leverage multicore architectures, guaranteed detection of attacks (not bypassable), secure monitor protection, and ability to precisely locate corrupted buffers, which is critical for testing, debugging, and security monitoring.

6.2 System Integrity

Existing OS integrity protection techniques can be broken down into three categories: (1) code integrity [45,57], (2) data integrity [5,63], (3) control flow integrity and control data integrity [1, 42, 76, 77]. Software cruising for data structure and object invariants in general falls into the third category. HookSafe [76] protects kernel hooks by relocating them to a dedicated page-aligned memory space. In contrast, software cruising does not do any hook relocating. The technique proposed by Petroni and Hicks [42] detects kernel control flow attacks by identifying persistent yet unexpected modifications of the kernel's CFG. It does not use any canaries. In contrast, software cruising can be applied to detect control flow attacks by comparing linkages between canaries with the linkages between the corresponding kernel data structures. Soft-Timer [77] uses soft timer interrupts while software cruising does not use any interrupt.

6.3 Information Flow Integrity

Security models for information flow controls were studied many years ago [17]. Recently, Decentralized Information Flow Control (DIFC) [19, 30, 34, 80] has attracted much interest. Compared to classic information flow control researches, which are model-oriented, DIFC is targeting pragmatic, system-oriented information flow control. DIFC projects have developed more practical and more usable declassification measures and information flow tracking (also called "taint tracking") mechanisms. Although taint tracking has been implemented in design-from-scratch DIFC systems such as HiStar [80], so far fine-grained information flow tracking still cannot be made practical in commodity software systems. This problem is a main motivation behind our plan to apply software cruising to information provenance and flow integrity checking.

6.4 Self-Healing Software

Self-healing (or self-fixing, self-repairing) software such as the Network Worm Vaccine Architecture [27, 43, 60], ClearView [41], and SHADOWS [59], aims to fix itself when something monitored goes wrong. However, runtime protection or monitoring mechanisms are often too expensive in practice to be applied in large scale. None of these solutions utilize non-blocking lock-free data structures and algorithms to reduce monitoring overhead. The software cruising technology can be combined with the self-healing software technology to make it more affordable since to be self-healing it has to be self-monitoring first!

7 Conclusion

We have presented a novel concurrent software monitoring technology, called *software cruising*. It leverages multicore architectures and utilizes *lock-free* data structures and algorithms to achieve efficient and scalable security monitoring. Applications include, but are not limited to, heap buffer integrity checking, kernel memory cruising, data structure and object invariant checking, rootkit detection, and information provenance and flow checking. In the software cruising framework, one or more dedicated threads, called cruising threads, are running concurrently with the monitored user or kernel code, to constantly check, or cruise, for security violations. We believe the software cruising technology would result in a game-changing capability in security monitoring for the cloud-based and traditional computing and network systems.

We have developed two prototypical systems: *Cruiser*, a lock-free concurrent heap buffer overflow monitor in user space, and *Kruiser*, a semi-synchronized non-blocking OS kernel cruiser. Cruiser is legacy code compatible and can be automatically applied to protect shrink-wrapped software and systems (source code or binary executables) transparently, and thus can gain extra security with virtually no cost for heap buffer overflow checking as StackGuard for stack buffers. Kruiser has a novel algorithm on concurrent, but *semi-synchronized non-blocking*, kernel heap integrity cruising. It is not fully synchronized, to reduce the performance overhead, but still ensures correctness regarding race conditions, deadlocks, and other typical concurrency issues.

Our preliminary results showed that software cruising can be deployed in practice with modest overhead. In user space, heap buffer overflow cruising incurs only 5 % performance overhead on average for the SPEC CPU2006 benchmark, and the Apache throughput slowdown is only 3 % maximum and negligible on average. In kernel space, it is negligible for SPEC, and 3.8 % for Apache. Both technologies can be deployed in large scale for (cloud) data centers and server farms in an automated manner.

Acknowledgements This research was supported in part by the National Science Foundation (NSF) under the grants CNS-1223710 and CNS-0905131, the Army Research Office (ARO) under the grant W911NF-09-1-0525 (MURI), and the Air Force Office of Scientific Research (AFOSR) under the grant W911NF1210055.

References

1. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity. In: Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS '05), pp. 340–353 (2005)
2. Akritidis, P., Costa, M., Castro, M., Hand, S.: Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors. In: USENIX Security '09, pp. 51–66 (2009)
3. Austin, T.M., Breach, S.E., Sohi, G.S.: Efficient detection of all pointer and array access errors. In: Proceedings of the ACM SIGPLAN conference on Programming language design and implementation, PLDI '04, pp. 290–301 (2004)
4. Avijit, K., Gupta, P.: Tied, libsafeplus, tools for runtime buffer overflow protection. In: USENIX Security '04, pp. 4–4 (2004)
5. Baliga, A., Ganapathy, V., Iftode, L.: Automatic inference and enforcement of kernel data structure invariants. In: ACSAC '08: Proceedings of the 2008 Annual Computer Security Applications Conference, pp. 77–86. IEEE Computer Society, Washington, DC, USA (2008). DOI http://dx.doi.org/10.1109/ACSAC.2008.29
6. Barrantes, E.G., Ackley, D.H., Palmer, T.S., Stefanovic, D., Zovi, D.D.: Randomized instruction set emulation to disrupt binary code injection attacks. In: Proceedings of the ACM conference on Computer and communications security, CCS '03, pp. 281–289 (2003)
7. Berger, E.D.: HeapShield: Library-based heap overflow protection for free. Tech. Report UMCS TR-2006-28, Univ. of Mass. Amherst (2006)
8. Berger, E.D., Zorn, B.G.: DieHard: probabilistic memory safety for unsafe languages. In: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation, PLDI '06, pp. 158–168. ACM, New York, NY, USA (2006). DOI http://doi.acm.org/10.1145/1133981.1134000. URL http://doi.acm.org/10.1145/1133981.1134000
9. Bhatkar, E., Duvarney, D.C., Sekar, R.: Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In: USENIX Security '03, pp. 105–120 (2003)
10. Castro, M., Costa, M., Harris, T.: Securing software by enforcing data-flow integrity. In: Proceedings of the 7th symposium on Operating systems design and implementation, OSDI '06, pp. 147–160. USENIX Association, Berkeley, CA, USA (2006). URL http://dl.acm.org/citation.cfm?id=1298455.1298470
11. Chatterjee, S., Lahiri, S., Qadeer, S., Rakamaric, Z.: A reachability predicate for analyzing low-level software. In: O. Grumberg, M. Huth (eds.) Proceedings of the 13th international conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07), *Lecture Notes in Computer Science*, vol. 4424, pp. 19–33. Springer Berlin Heidelberg (2007). DOI 10.1007/978-3-540-71209-1_4. URL http://dx.doi.org/10.1007/978-3-540-71209-1_4
12. Chiueh, T.C., Hsu, F.H.: RAD: A compile-time solution to buffer overflow attacks. In: Proceedings of the The 21st International Conference on Distributed Computing Systems (ICDCS '01), pp. 409–417 (2001)
13. Condit, J., Hackett, B., Lahiri, S.K., Qadeer, S.: Unifying type checking and property checking for low-level code. In: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '09, pp. 302–314. ACM, New York, NY, USA (2009). DOI http://doi.acm.org/10.1145/1480881.1480921. URL http://doi.acm.org/10.1145/1480881.1480921

14. Cowan, C., Beattie, S.: PointGuard: protecting pointers from buffer overflow vulnerabilities. In: USENIX Security '03, pp. 91–104 (2003)
15. Cowan, C., Pu, C.: StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In: USENIX Security '98, pp. 63–78 (1998)
16. Cox, B., Evans, D., Filipi, A., Rowanhill, J., Hu, W., Davidson, J., Knight, J., Nguyen-Tuong, A., Hiser, J.: N-variant systems: a secretless framework for security through diversity. In: USENIX Security '06, pp. 105–120 (2006)
17. Denning, D.: A lattice model of secure information flow. Communications of the ACM **19**(5), 236–243 (1976)
18. Dor, N., Rodeh, M., Sagiv, M.: CSSV: towards a realistic tool for statically detecting all buffer overflows in C. In: Proceedings of the ACM SIGPLAN conference on Programming language design and implementation, PLDI '03, pp. 155–167 (2003)
19. Efstathopoulos, P., Krohn, M., VanDeBogart, S., Frey, C., Ziegler, D., Kohler, E., Mazieres, D., Kaashoek, F., Morris, R.: Labels and event processes in the Asbestos operating system. In: Proceedings of the Nineteenth ACM SIGOPS symposium on Operating systems principles, SOSP '05 (2005)
20. Electric Fence: Malloc debugger. Http://directory.fsf.org/project/ElectricFence/
21. Frantzen, M., Shuey, M.: StackGhost: Hardware facilitated stack protection. In: USENIX Security '01, pp. 55–66 (2001)
22. Hastings, R., Joyce, B.: Purify: Fast detection of memory leaks and access errors. In: the Winter 1992 Usenix Conference, pp. 125–136 (1992)
23. Hofmeyr, S.A., Forrest, S., Somayaji, A.: Intrusion detection using sequences of system calls. Journal of Computer Security **6**(3), 151–180 (1998). URL http://dl.acm.org/citation.cfm?id=1298081.1298084
24. IBM: ProPolice detector. Http://www.trl.ibm.com/projects/security/ssp/
25. IBM System/370 Extended Architecture, Principles of Operations: IBM Publication No. SA22-7085 (1983)
26. Jim, T., Morrisett, J.G., Grossman, D., Hicks, M.W., Cheney, J., Wang, Y.: Cyclone: A safe dialect of C. In: USENIX Annual Technical Conference (ATC '02), pp. 275–288 (2002)
27. Keromytis, A.D.: The case for self-healing software. In: Aspects of Network and Information Security: Proceedings NATO Advanced Studies Institute (ASI) on Network Security and Intrusion Detection (2005)
28. King, S.T., Chen, P.M.: Backtracking intrusions. In: Proceedings of the nineteenth ACM symposium on Operating systems principles, SOSP '03, pp. 223–236. ACM, New York, NY, USA (2003). DOI 10.1145/945445.945467. URL http://doi.acm.org/10.1145/945445.945467
29. Kiriansky, V., Bruening, D., Amarasinghe, S.P.: Secure execution via program shepherding. In: USENIX Security '02, pp. 191–206 (2002)
30. Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M.F., Kohler, E., Morris, R.: Information flow control for standard OS abstractions. In: Proceedings of the twenty-first ACM SIGOPS symposium on Operating systems principles, SOSP (2007)
31. Lahiri, S.K., Qadeer, S.: Verifying properties of well-founded linked lists. In: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '06, pp. 115–126. ACM, New York, NY, USA (2006). DOI http://doi.acm.org/10.1145/1111037.1111048. URL http://doi.acm.org/10.1145/1111037.1111048
32. Lamport, L.: Proving the correctness of multiprocess programs. IEEE Trans. Softw. Eng. **3**(2), 125–143 (1977)
33. Michael, M.M.: Hazard pointers: Safe memory reclamation for lock-free objects. IEEE Trans. Parallel Distrib. Syst. **15**(6), 491–504 (2004)
34. Myers, A., Liskov, B.: Protecting privacy using the decentralized label model. ACM Transactions on Computer Systems (2000)

35. Myers, A.C., Liskov, B.: A decentralized model for information flow control. In: Proceedings of the sixteenth ACM symposium on Operating systems principles, SOSP '97, pp. 129–142. ACM, New York, NY, USA (1997). DOI 10.1145/268998.266669. URL http://doi.acm.org/10.1145/268998.266669
36. Necula, G.C., Condit, J., Harren, M., McPeak, S., Weimer, W.: CCured: type-safe retrofitting of legacy software. ACM Trans. Program. Lang. Syst. **27**(3), 477–526 (2005)
37. Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature-generation of exploits on commodity software. In: Proceedings of the Network and Distributed System Security Symposium (NDSS '05) (2005)
38. NIST. SAMATE Reference Dataset: Http://samate.nist.gov/SRD
39. Novark, G., Berger, E.D.: DieHarder: securing the heap. In: Proceedings of the 17th ACM conference on Computer and communications security, CCS '10, pp. 573–584. ACM, New York, NY, USA (2010). DOI http://doi.acm.org/10.1145/1866307.1866371. URL http://doi.acm.org/10.1145/1866307.1866371
40. Ou, X., Govindavajhala, S., Appel, A.W.: MulVAL: a logic-based network security analyzer. In: Proceedings of the 14th conference on USENIX Security Symposium - Volume 14, pp. 113–128. USENIX Association, Berkeley, CA, USA (2005). URL http://dl.acm.org/citation.cfm?id=1251398.1251406
41. Perkins, J.H., Kim, S., Larsen, S., Amarasinghe, S., Bachrach, J., Carbin, M., Pacheco, C., Sherwood, F., Sidiroglou, S., Sullivan, G., Wong, W.F., Zibin, Y., Ernst, M.D., Rinard, M.: Automatically patching errors in deployed software. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, SOSP '09, pp. 87–102. ACM, New York, NY, USA (2009). DOI http://doi.acm.org/10.1145/1629575.1629585. URL http://doi.acm.org/10.1145/1629575.1629585
42. Petroni Jr., N.L., Hicks, M.: Automated detection of persistent kernel control-flow attacks. In: Proceedings of the 14th ACM conference on Computer and communications security, CCS '07, pp. 103–115 (2007)
43. Portokalidis, G., Keromytis, A.D.: REASSURE: A self-contained mechanism for healing software using rescue points. In: Advances in Information and Computer Security—6th International Workshop, IWSEC 2011, Tokyo, Japan, November 8–10, 2011. Proceedings, *Lecture Notes in Computer Science*, vol. 7038, pp. 16–32. Springer (2011)
44. Prasad, M., Chiueh, T.C.: A binary rewriting defense against stack based buffer overflow attacks. In: Usenix Annual Technical Conference (Usenix ATC '03), pp. 211–224 (2003)
45. Riley, R., Jiang, X., Xu, D.: Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing. In: Proceedings of the 11th international conference on Recent advances in intrusion detection, RAID '08 (2008)
46. Robertson, W., Kruegel, C., Mutz, D., Valeur, F.: Run-time detection of heap-based overflows. In: Proceedings of the 17th Usenix Conference on System Administration (LISA '03), pp. 51–60. Usenix Association, Berkeley, CA, USA (2003)
47. Roethlisberge, D.: Omnikey Cardman 4040 Linux driver buffer overflow (2007). http://www.securiteam.com/unixfocus/5CP0D0AKUA.html
48. Ruwase, O., Lam, M.S.: A practical dynamic buffer overflow detector. In: Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS '04), pp. 159–169 (2004)
49. Salamat, B., Jackson, T., Gal, A., Franz, M.: Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In: Proceedings of the 4th ACM European conference on Computer systems (EuroSys '09), pp. 33–46 (2009)
50. Schneider, F.: Blueprint for a science of cybersecurity. The Next Wave **19**(2), 47–57 (2012)
51. SecurityFocus: Wu-ftpd file globbing heap corruption (2001). http://www.securityfocus.com/bid/3581
52. SecurityFocus: Sudo password prompt heap overflow (2002). http://www.securityfocus.com/bid/4593
53. SecurityFocus: CVS directory request double free heap corruption (2003). http://www.securityfocus.com/bid/6650

54. SecurityFocus: Mozilla Firefox and Seamonkey regular expression parsing heap buffer overflow (2009). http://www.securityfocus.com/bid/35891
55. SecurityFocus: libHX 'HX_split()' remote heap-based buffer overflow (2010). http://www.securityfocus.com/bid/42592
56. SecurityFocus: Lynx browser 'convert_to_idna()' function remote heap based buffer overflow (2010). http://www.securityfocus.com/bid/42316
57. Seshadri, A., Luk, M., Qu, N., Perrig, A.: SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In: Proceedings of the twenty-first ACM SIGOPS symposium on Operating systems principles, SOSP '07, pp. 335–350 (2007)
58. Sharif, M.I., Lee, W., Cui, W., Lanzi, A.: Secure in-VM monitoring using hardware virtualization. In: Proceedings of the 16th ACM conference on Computer and communications security, CCS '09, pp. 477–487 (2009)
59. Shehory, O.: SHADOWS: Self-healing complex software systems. In: Automated Software Engineering, pp. 71–76 (2008). DOI 10.1109/ASEW.2008.4686296
60. Sidiroglou, S., Laadan, O., Perez, C., Viennot, N., Nieh, J., Keromytis, A.D.: ASSURE: automatic software self-healing using rescue points. In: M.L. Soffa, M.J. Irwin (eds.) ASPLOS, pp. 37–48. ACM (2009)
61. Solar Designer: Non-executable user stack (1997). Http://www.openwall.com/linux/
62. sqrkkyu, twzi: Attacking the core: Kernel exploiting notes (2007). Http://phrack.org/issues.html
63. Srivastava, A., Erete, I., Giffin, J.: Kernel data integrity protection via memory access control. Tech. Rep. GT-CS-09-04, Georgia Institute of Technology (2009)
64. StackShield: (2000). Http://www.angelfire.com/sk/stackshield/
65. The PaX project: Http://pax.grsecurity.net/
66. Tian, D., Zeng, Q., Wu, D., Liu, P., Hu, C.: Kruiser: Semi-synchronized non-blocking concurrent kernel heap buffer overflow monitoring. In: Proceedings of the 19th Network and Distributed System Security Symposium, NDSS '12 (2012)
67. Tiwari, M., Wassel, H.M., Mazloom, B., Mysore, S., Chong, F.T., Sherwood, T.: Complete information flow tracking from the gates up. In: Proceedings of the 14th international conference on Architectural support for programming languages and operating systems, ASPLOS XIV, pp. 109–120. ACM, New York, NY, USA (2009). DOI 10.1145/1508244.1508258. URL http://doi.acm.org/10.1145/1508244.1508258
68. Tsai, T.K., Singh, N.: Libsafe: Transparent system-wide protection against buffer overflow attacks. In: Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN '02), pp. 541–541 (2002)
69. US-CERT/NIST: CVE-2008-1673. Http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-1673
70. US-CERT/NIST: CVE-2009-2407. Http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-2407
71. US-CERT/NIST: National vulnerability database, CVE-2002-0392. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2002-0392
72. US-CERT/NIST: National vulnerability database, CVE-2003-0252. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2003-0252
73. Valgrind: Http://valgrind.org/
74. Wagner, D., Foster, J.S., Brewer, E.A., Aiken, A.: A first step towards automated detection of buffer overrun vulnerabilities. In: Proceedings of the 7th Network and Distributed System Security Symposium, NDSS '00, pp. 3–17 (2000)
75. Wahbe, R., Lucco, S., Anderson, T.E., Graham, S.L.: Efficient software-based fault isolation. In: Proceedings of the fourteenth ACM symposium on Operating systems principles, SOSP '93, pp. 203–216. ACM, New York, NY, USA (1993). DOI 10.1145/168619.168635. URL http://doi.acm.org/10.1145/168619.168635
76. Wang, Z., Jiang, X., Cui, W., Ning, P.: Countering kernel rootkits with lightweight hook protection. In: CCS '09: Proceedings of the 16th ACM Conference on Computer and Communications Security (2009)

77. Wei, J., Payne, B.D., Giffin, J., Pu, C.: Soft-timer driven transient kernel control flow attacks and defense. In: ACSAC '08: Proceedings of the 2008 Annual Computer Security Applications Conference, pp. 97–107. IEEE Computer Society, Washington, DC, USA (2008). DOI http://dx.doi.org/10.1109/ACSAC.2008.40
78. Xiong, X., Tian, D., Liu, P.: Practical protection of kernel integrity for commodity OS from untrusted extensions. In: Proceedings of the Network and Distributed System Security Symposium, NDSS '11. The Internet Society (2011)
79. Xu, J., Kalbarczyk, Z., Patel, S., Iyer, R.: Architecture support for defending against buffer overflow attacks. In: Workshop Evaluating & Architecting Sys. Depend. (2002)
80. Zeldovich, N., Boyd-Wickizer, S., Kohler, E., Mazieres, D.: Making information flow explicit in HiStar. Communications of the ACM (2011)
81. Zeldovich, N., Kannan, H., Dalton, M., Kozyrakis, C.: Hardware enforcement of application security policies using tagged memory. In: Proceedings of the 8th USENIX conference on Operating systems design and implementation, OSDI'08, pp. 225–240. USENIX Association, Berkeley, CA, USA (2008). URL http://dl.acm.org/citation.cfm?id=1855741.1855757
82. Zeng, Q., Wu, D., Liu, P.: 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, PLDI '11, pp. 367–377. ACM, New York, NY, USA (2011). DOI http://doi.acm.org/10.1145/1993498.1993541. URL http://doi.acm.org/10.1145/1993498.1993541