Privacy-preserving Virtual Machine Checkpointing Mechanism [a]

Mikhail I. Gofman

Computer Science Dept., California State Univ. at Fullerton, USA
E-mail:mgofman@ecs.fullerton.edu

Ruiqi Luo

Computer Science Dept., State Univ. of New York at Binghamton, USA
E-mail:rluo1@binghamton.edu

Chad Wyszynski

Computer Science Dept., California State Univ. at Fullerton, USA
E-mail:chad.wyszynski@csu.fullerton.edu

Yaohui Hu, Ping Yang*, Kartik Gopalan

Computer Science Dept., State Univ. of New York at Binghamton, USA
E-mail:{yhu15,pyang,kartik}@binghamton.edu
*Corresponding author

Abstract: Virtual Machines (VMs) have been widely adopted in cloud platforms to improve server consolidation and reduce operating costs. VM checkpointing is used to capture a persistent snapshot of a running VM and to later restore the VM to a previous state. Although VM checkpointing eases system administration, such as in recovering from a VM crash or undoing a previous VM activity, it can also increase the risk of exposing users' confidential data. This is because the checkpoint may store a VM's physical memory pages and disk contents that contain confidential data such as clear text passwords and credit card numbers.

This paper presents the design and implementation of *SPARC*, a Security and Privacy AwaRe virtual machine Checkpointing mechanism. *SPARC* enables users to selectively exclude users' confidential data within a VM from being checkpointed. We describe the design challenges in effectively tracking and excluding process-specific memory and disk contents from the checkpoint file for a VM running on the commodity Linux operating system. We also present techniques to track process dependencies due to inter-process communication and to account for such dependencies in *SPARC* .

Keywords: Virtual Machine Checkpointing; Privacy; Security.

Biographical notes: Mikhail Gofman is an Assistant Professor at California State University at Fullerton. He received his PhD in Computer Science at State

University of New York at Binghamton. His research interests include security, privacy, access control, security policy analysis, and biometrics.

Ruiqi Luo received his PhD in Computer Science at State University of New York at Binghamton.

Chad Wyszynski is a master student at California State University at Fullerton.

Yaohui Hu is a PhD student at State University of New York at Binghamton.

Ping Yang is an Assistant Professor at State University of New York at Binghamton. She received her PhD in Computer Science at State University of New York at Stony Brook. Her research interests include security, privacy, virtualization, access control, security policy analysis, and formal methods.

Kartik Gopalan is an Associate Professor at State University of New York at Binghamton. He received his PhD in Computer Science at State University of New York at Stony Brook. His research interests include virtualization, operating systems, distributed systems, networks, security, and privacy.

1 Introduction

Virtualization technology is being widely adopted in grid and cloud computing platforms to improve server consolidation and reduce operating costs. On one hand, virtual machines (VMs) help improve security in the cloud computing infrastructure through greater isolation and more transparent malware analysis and intrusion detection (e.g. Nguyen et al. (2009); Oliveira and Wu (2009); Riley et al. (2008); Dinaburg et al. (2008); Dunlap et al. (2002); Garfinkel and Rosenblum (2003); Joshi et al. (2005); Seshadri et al. (2007); Payne et al. (2008); Kourai and Chiba (2005)). On the other hand, virtualization also gives rise to new challenges in maintaining security and privacy in cloud computing infrastructures. Although significant advances have been made in developing techniques to secure the execution of VMs, a number of challenges remain unaddressed. In this paper, we present techniques to address some of the security and privacy issues in VM checkpointing.

VM checkpointing saves a persistent snapshot (or a checkpoint) of the entire memory and disk state of a VM in execution, which can be later used for various purposes such as restoring the VM to a previous state, recovering a long-running process after a crash, distributing a VM image with a preset execution state among multiple users, archiving a VM's execution record, conducting forensic examination, etc. Most hypervisors such as VMware, Hyper-V, VirtualBox, KVM, and Xen, and some of the cloud platforms such as Amazon EC2, support VM memory and/or disk checkpointing.

Despite its many benefits, VM checkpointing also has its drawbacks from a security perspective. Checkpoints are stored on persistent storage and contain the VM's physical memory and disk contents at a given time instant, and hence can drastically prolong the

* This article is a revised and expanded version of a paper entitled "SPARC: A Security and Privacy Aware Virtual Machine Checkpointing Mechanism" presented at the 10th annual ACM Workshop on Privacy in the Electronic Society (WPES), in conjunction with the ACM Conference on Computer and Communications Security (CCS), Chicago, USA, Oct. 17–21, 2011.

lifetime of confidential data stored in memory and disk, such as clear text passwords, credit card numbers, and other confidential data which would normally be quickly discarded after usage.

We have demonstrated the vulnerability of VM checkpointing using a common scenario of entering credit card information in a website. As shown in Figure 1, we started the FireFox browser inside a VirtualBox VM. We then connected to http://www.amazon.com, clicked "my account" to add credit card information, entered 9149239648 in the credit card number field, and then performed checkpointing. When searching through the checkpoint file with a hex editor, we were able to locate the credit card number we had entered earlier. In some of our experiments, the checkpoint file contains the string "addCreditCardNumber=9149239648", which can enable an attacker to locate the credit card number easily by searching for the string "CreditCard" in the checkpoint. Furthermore, we found that even if the checkpointing is performed *after* the browser terminates, the credit card number can still be located in the checkpoint file, likely because the browser's memory was not cleared after the browser terminated. In other words, the common advice to "close your browser after logging out" may give users a false sense of security. *Many users are not aware that their input data may still reside in memory even after the application that has processed such data terminates.* Such users may mistakenly assume that checkpointing the VM is safe simply because the application has terminated.

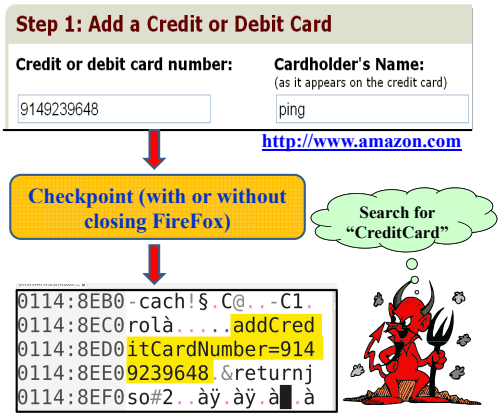


Figure 1 A scenario where the credit card number is checkpointed.

Besides memory, even checkpointing a VM's disk may also store users' confidential data in the snapshot. For example, Balduzzi et al. (2012) analyzed 5303 public Amazon EC2 snapshots and found that many of them contain sensitive information such as passwords, browser history, and deleted files.

Previous work on minimizing the data lifetime (e.g. Garfinkel et al. (2004); Chow et al. (2005)) has primarily focused on clearing the deallocated memory. However, this does not prevent memory pages from being checkpointed before they are deallocated. Garfinkel and Rosenblum (2005) and VMware ace proposed to protect the checkpointed information by encrypting the checkpoint files. Encrypting the checkpoint can help protect confidential data if the data is needed after the restoration of the VM. However, encrypting the checkpoint is not suitable to protect confidential data that should be quickly discarded after its use due to the following reasons: (1) it still prolongs the lifetime of confidential data that should normally be quickly destroyed after use, (2) when a checkpointed VM is restored, the confidential data will be decrypted and loaded into the memory of the VM, thus making it vulnerable again, and (3) the checkpoint may be shared among multiple users (for example, among multiple programmers for the purpose of software development and debugging) and encryption does not prevent users who share the checkpoint from accessing the confidential

data. It is also insufficient to encrypt just the memory containing the confidential data because VM checkpointing can occur just when a program decrypts the data.

In this paper, we present *SPARC*, a Security and Privacy AwaRe virtual machine Checkpointing mechanism, which enables users to exclude applications and disk contents that contain users' confidential data from being checkpointed without affecting the current execution of applications. Our main contributions are summarized below.

- We have developed techniques for excluding applications and terminals that may process users' confidential data from being checkpointed. We have implemented a prototype based on the VirtualBox 3.1.2_OSE hypervisor and Ubuntu Linux 9.10 guest. Our experimental results show that, our techniques impose only $1\% - 5.3\%$ checkpointing overhead with common application workloads, when a single process is excluded from memory checkpointing.
- We have developed techniques to track process dependencies due to inter-process communication by using hooks in the guest kernel. *SPARC* accounts for such dependencies during VM checkpointing and restoration to prevent the leakage of confidential user data and to maintain system stability. Our experimental results show that, it takes 0.01 - 0.52 millisecond to detect one dependency.
- We have developed techniques to identify disk contents that may contain confidential data and to exclude them from being checkpointed.

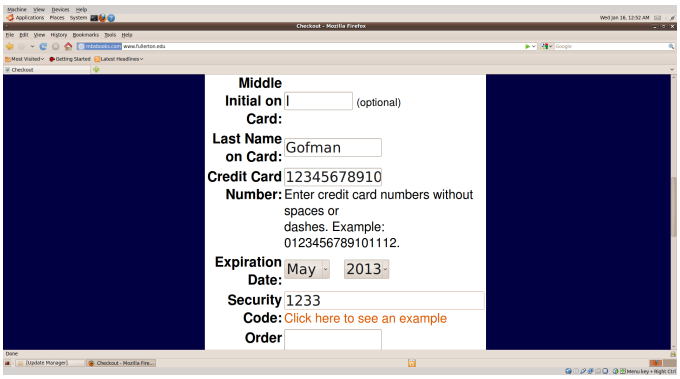
Organization

The rest of this chapter is organized as follows. Section 2 provides a threat model. Section 3 presents techniques for excluding the entire memory footprint of user applications and terminal processes from being checkpointed and presents the experimental results. Section 4 describes techniques for excluding disk contents that may store confidential data from being checkpointed. Section 5 gives the related work. Section 6 concludes the paper.

2 Assumptions and Threat Model

Our proposed research is based on the following assumptions and threat model. We assume that the hypervisor, the guest OS in the target VM, the checkpointer, and user applications are not compromised before and during checkpointing. The checkpoint may be stored on the physical machine on which the VM is checkpointed or stored on a remote storage. We also assume that anytime after the VM is checkpointed, the attacker may gain access to the checkpoint, either by compromising the storage containing the checkpoint, or because the checkpoint was made publicly available. In order to obtain the confidential data stored in the checkpoint, the attacker may subsequently study the checkpoint, copy the checkpoint to a remote machine for later analysis, or restore the VM from the checkpoint. The attacker may also modify the checkpoint, delete the checkpoint, send the checkpoint to other unauthorized users, or make the checkpoint publicly available.

In this paper, we consider the inter-process communication within the same VM, but not that among different VMs or different physical machines. For example, we do not consider the case where the confidential data is sent from one physical machine to a VM on another physical machine, which is then checkpointed.



(a)

(b)

Figure 2 VM restored using (a) VirtualBox's default checkpointing mechanism; and (b) *SPARC* with FireFox excluded.

3 Privacy-Aware Virtual Machine Memory Checkpointing

3.1 Excluding A Single Process and Terminal

This section presents techniques for excluding a single application as well as the terminal on which the application is running from being checkpointed. The key idea is to identify memory pages of the application and the terminal that store confidential data and to exclude them from being checkpointed. The key challenge is to exhaustively exclude all memory locations in the VM that contain confidential data, while at the same time, ensuring that the VM's stability and consistency are maintained when the VM is restored from the checkpoint.

3.1.1 Excluding a Single Application from Being Checkpointed

This section presents the memory checkpointing mechanism in *SPARC* , which enables users to exclude memory pages of applications that may process confidential data (e.g. FireFox, Internet Explorer, Email clients, etc) from being checkpointed. Although our techniques are presented in the context of VirtualBox memory checkpointing, they are applicable to all hypervisors.

Consider an example where a user has entered a credit card number into the FireFox web browser as shown in Figure 2(a). If the user performs checkpointing after the credit card number is entered, then the credit card number will be stored in the checkpoint.

Figure 2(b) gives the screenshot of the VM restored using *SPARC* in which FireFox and the data processed by FireFox are excluded from being checkpointed. Note that *SPARC* will not affect the current execution of FireFox since the corresponding memory pages are not cleared from the RAM of the executing VM.

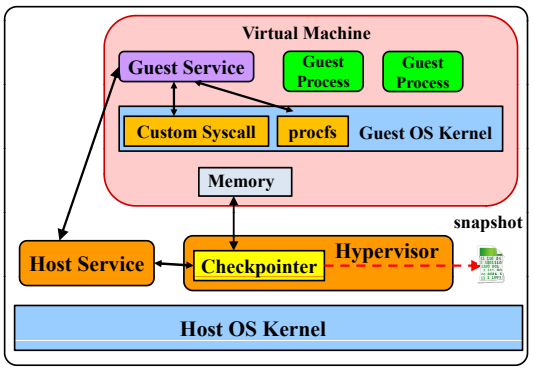


Figure 3 The architecture of memory checkpointing in *SPARC* .

Figure 3 gives the high-level architecture of *SPARC*. First, the user selects an application that he or she wishes to exclude from being checkpointed. Next, a special process called the *guest service* inside the VM collects physical addresses of memory pages that belong to the application being excluded from the VM checkpoint. When checkpointing is initiated, a process in the host system, called the *host service*, requests the guest service to provide the physical addresses of memory pages that need to be excluded. The host service then relays the addresses to *the checkpointer* in the hypervisor, which in turn clears the specified pages in the checkpoint file. The pseudocode illustrating our privacy-preserving checkpointing mechanism is given in Figure 4.

Excluding process physical memory

Below, we describe how *SPARC* identifies memory pages that belong to a process with ID `pid` and excludes those pages from being checkpointed. First, the guest service invokes a system call [a] that locates the process descriptor (i.e. `struct task_struct`) associated with the process, which links together all information of a process such as memory, opened files, associated terminal, etc.. From the process descriptor, we obtain a memory descriptor (`struct mm_struct`) which contains the starting and ending addresses of each segment (e.g. program code segment, heap segment, stack segment etc.) of the process virtual memory.

Next, the guest service breaks up the memory of each segment into its constituent virtual pages and converts the virtual address of each page into the physical address based on file `/proc/pid/pagemap` in the *process file system* (`procfs`), which is a virtual file system that enables access and modification of kernel parameters from the user space through a file-like interface. To avoid affecting other processes in the system, we do not exclude resident pages that are being mapped more than once; such pages can be identified by checking the file `/proc/kpagecount` that records the number of times each physical page has been mapped. We also do not exclude segments representing executable images because such segments do not contain confidential data and clearing them may affect processes that share the same in-memory executable image. In addition, we skip over segments representing memory-mapped files (e.g. libraries) because clearing such segments may affect processes that map the same files into their memory.

[a] System calls have been used for ease of prototyping and can be easily replaced with a more transparent and extensible *ioctl* interface.

```
priv_checkpoint(proc){
    the guest service freezes all user-space processes in the guest OS except the guest
service;
    the guest service collects the set of virtual addresses va of all memory pages
belonging to the confidential process proc;
    the guest service converts va to the corresponding guest physical addresses pa;
    the guest service sends pa to the host service;
    the host service relays pa to the checkpointer;
    for every page P in the guest VM {
        ad = guest physical address of page P;
        if (ad ∉ pa) {checkpoint page P;}
        else{checkpoint a page containing zeros;}
    }
    the guest service unfreezes all user-space processes in the guest OS;
}
```

Figure 4 The algorithm for memory checkpointing in *SPARC* .

Finally, the guest service sends physical addresses of memory pages that need to be excluded from the checkpoint to the host service, which in turn relays the addresses to the hypervisor. Prior to saving a physical page to the checkpoint file, the hypervisor checks if the guest physical address of the page matches one of the received addresses. If so, it saves a page containing all 0's. Otherwise, it saves the content of the page. This is done by modifying the function `pgmSavePages()` in VirtualBox. Further, since memory pages are constantly swapped between the disk and the physical memory, the virtual-to-physical memory mappings of a process may change after the physical addresses are collected. We overcome this problem by freezing all user space processes except the guest service prior to gathering physical addresses, using the *freezer subsystem* of the guest kernel. When the VM is restored, the guest service detects the restoration event and sends a `SIGKILL` signal to processes whose memory contents were excluded during checkpointing. The `SIGKILL` signal enables the guest kernel to clean up any residual state (other than memory) for the excluded processes before the VM resumes. The guest service then unfreezes the all processes and the execution proceeds as normal. In addition, we have also modified kernel functions to clear deallocated memory pages belonging to the process prior to deallocation.

Excluding pages of a process in the page cache

Page cache is used by the kernel to speed up disk operations by caching disk data in the main memory. If an application performs disk I/O operations, the confidential data processed by the application may reside in the page cache. For example, we found that, when searching for any string using the Google search engine through FireFox, the string appears in the kernel's page cache, possibly because Google caches suggestions for frequent searches on the local disk. Moreover, when a process terminates, the page cache may retain some of the pages of the terminated process for a period of time in case that the same data is accessed by another process in the near future.

SPARC excludes the cached pages of a process in the checkpoint as follows. First, it retrieves the file descriptor table (`struct file ** fd`) from the process descriptor of the process, which comprises all file descriptors belonging to the process. Next, for each file descriptor that represents an open file, it obtains information about pages that cache data

from the field `struct address_space i_mapping`. Finally, it uses this structure to obtain page descriptors representing pages in the page cache, converts the page descriptors to physical addresses of the pages, transfers the addresses to the host service, and clears them.

Note that when a process closes a file descriptor, the descriptor is removed from the file descriptor table of the process. As a result, if the process closes the descriptor prior to the checkpointing, the above approach will fail to detect the associated pages in the page cache. To counter this problem, whenever a file descriptor is closed, we evict and clear all pages stored in the page cache associated with the closed file descriptor. In addition, the (cleared) pages in the page cache may also be used by other processes. To avoid affecting processes that rely on such pages, when the VM is restored (but before the processes are thawed), we flush all pages used by the processes from the page cache.

Excluding pipe buffers

Pipes and FIFOs are commonly used for implementing producer/consumer relationship between two processes. For example, the shell program makes use of pipes to connect output of one process to the input of another through e.g. command "ls | grep myfile". FireFox also uses pipes to trace `malloc()` memory allocations. FIFOs are similar to pipes but allow communication of two unrelated processes. For example, in a terminal, a user can create a FIFO called `myfifo` with command `mkfifo myfifo`. Issuing command `echo "Data lifetime is important" > myfifo` will write the string "Data lifetime is important" to the buffer of `myfifo`. Subsequent command `cat myfifo` will remove the string from the buffer of `myfifo` and print "Data lifetime is important" to the terminal. FIFOs are frequently used by Google Chrome to implement communications between the renderer process and the browser process.

Data exchanged via pipes and FIFOs flows through a *pipe buffer* in the kernel. As a result, we need to sanitize pipe buffers used by the process that needs to be excluded. The pipe buffers are sanitized as follows. First, we locate the file descriptors opened by the process that represent pipes and FIFOs. We then retrieve the associated pipe buffer descriptors (of type `struct pipe_buffer`) from each file descriptor. Finally, we retrieve the descriptors of pages storing inter-process data from each pipe buffer descriptor and convert then into the physical addresses of pages they represent.

Excluding socket buffers

The kernel associates each socket with a list of socket buffers (of type `struct sk_buff`), which contain data exchanged over the socket. If a process sends or receives confidential data via an open socket (e.g. through `read()` and `write()` system calls), the data may be stored in the `sk_buff` structure of the sockets used by the process. Therefore, when excluding a process, we also detect all sockets opened by the process and sanitize the memory pages associated with the `sk_buff` structure of the sockets.

Identifying file descriptors of a process that represent sockets is similar to detecting pipes and FIFOs. First, we retrieve the socket descriptor (stored in `struct socket`) from each file descriptor that represents a socket. Each socket descriptor contains structure (`struct sock`) that encapsulates network layer representation of the socket. This structure consists of the queue of socket buffers that are ready to be sent via socket (`struct sk_buff_head sk_write_queue`) or have been received via socket (`struct`

`sk_buff_head sk_receive_queue`). We then traverse the queue to determine the physical memory address of each socket buffer that stores the data.

Preventing privacy leakage through screen display after VM restoration

If the excluded process displays confidential information, such as credit card number, on the screen when checkpointing is performed, then the confidential information may display on the screen for a brief moment, when the VM is restored and before the process is terminated. To address the problem, during checkpointing, we invoke the `XCreateWindow()` API provided by X-Windows to visually cover the windows of the excluded process with black rectangles. When the checkpointing completes, the rectangles are removed and the user continues to use the VM. When the VM is restored, the window is removed briefly after sending the `SIGKILL` signals to the excluded process and unfreezing the process.

3.1.2 Excluding Terminal Applications

Applications running on terminals may display confidential data on the terminal. As a result, we also need to exclude terminals on which the excluded applications are running. In Linux, there are two main types of terminals: *virtual consoles* and *pseudo terminals*. Typically, a Linux system contains 7 virtual consoles named `tty1-tty7`. The first 6 consoles provide a text terminal interface consisting of the login and shell, and the 7th console provides a graphical interface. The pseudo terminal emulates a text terminal within some other graphical system. A typical pseudo terminal application such as *xterm* forks off a shell process (e.g. `bash`). When the user runs a command (e.g. `ls`) on a terminal, the shell forks off a child process and replaces the child's executable image with the code of the specified command.

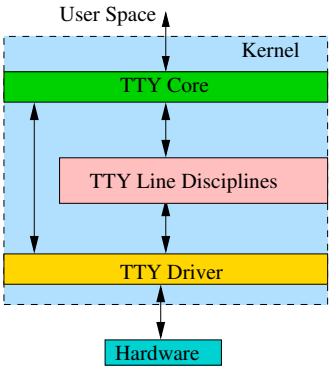


Figure 5 Teletype (TTY) subsystem architecture

All terminals rely on the Teletype (TTY) subsystem in the kernel. Figure 5 gives the architecture of the TTY subsystem in which arrows specify the flow of data. The uppermost layer of the TTY subsystem is *TTY core*, which arbitrates the flow of data between user space and TTY. The data received by the TTY core is sent to *TTY line disciplines*, which convert data to a protocol specific format such as PPP or Bluetooth. Finally, the data is sent to the *TTY driver*, which converts the data to the hardware specific format and sends it to the hardware. All data received by the TTY driver from the hardware flows back up to the line disciplines and finally to the TTY core where it can be retrieved from the user space. Sometimes the TTY core and the TTY driver communicate directly (Corbet et al. (2005)).

Identifying Terminal where a Process is Running

The kernel associates each process descriptor with a TTY structure (`tty_struct`), which links together all information relevant to the instance of the TTY subsystem associated with the process. We can determine the terminal on which a process is running by examining the `name` field of the TTY structure. The name of the terminal is either $ttyxx$ (if the terminal is a virtual console) or $ptsxx$ (if the terminal is a pseudo terminal).

Once we identify the name of the terminal on which the process that needs to be excluded is running, we traverse all process descriptors to detect all other processes that are running on the same terminal and exclude them from being checkpointed. If the process is running on a pseudo terminal, we also exclude the pseudo terminal application (e.g. `xterm`) as it may contain the output of the process. The terminal application is usually not attached to the same terminal as the process being excluded. However, it can be detected by following the `task_struct * real_parent` field in the descriptor of the process running on the terminal, which points to the process descriptor of the parent process, until the descriptor of the terminal application is reached. The terminal application and all its descendants are then excluded as described in Section 3.1.1.

Excluding TTY Information

We sanitize the TTY subsystem associated with the terminal by clearing buffers used at each level of the TTY subsystem. TTY core uses specialized buffers (of type `struct tty_buffer`) to store information received from the user space. TTY line disciplines use three buffers to store the data received from the TTY driver (`read_buf`), the data received from the TTY core that needs to be written to the TTY device (`write_buf`), and characters received from the device that need to be echoed back to the device (`echo_buf`). [b]

The virtual console is excluded as follows. The kernel maintains an array of structures (`struct vc vc_cons[]`), representing available virtual consoles. We identify the target console by traversing this array and comparing the number of each console against the number of the target console. We then use the identified console structure to access the TTY subsystem associated with the console, and clear the memory of all buffers of the TTY subsystem. Finally, we obtain the physical addresses of the TTY buffers and send the addresses along with buffer sizes to the host service. In our experiments, we did not find any information buffered in the console driver.

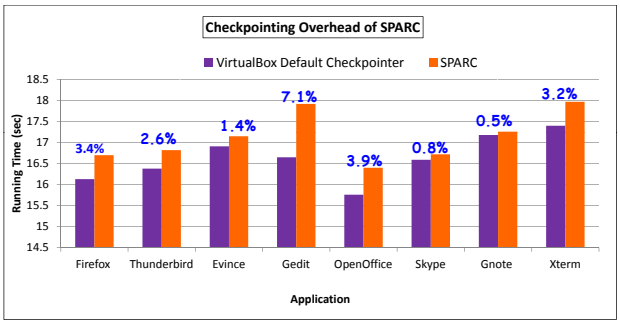
Excluding pseudo terminals is slightly more complex than excluding virtual consoles because we also need to sanitize the pseudo terminal driver. The pseudo terminal driver consists of two cooperating virtual character devices: *pseudo terminal master (*`ptm`*)* and *pseudo terminal slave (*`pts`*)*. Data written to `ptm` is readable from `pts` and vice-versa. Therefore, in a terminal emulator, a parent process can open the `ptm` and control the I/O of its child processes that use the `pts` end as their terminal device i.e. `stdin`, `stdout`, and `stderr` streams. After the TTY subsystem instances of both devices are identified, the rest of the operations are similar to operations involved in excluding a virtual console.

In addition, sensitive data may linger in the TTY subsystem buffers even after they are deallocated. Therefore, we modify functions that deallocate such buffers to clear them prior to deallocation.

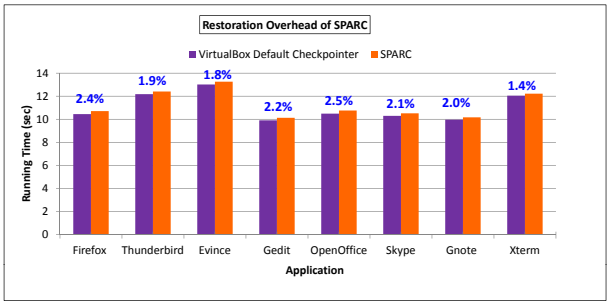
Experiments

We have performed the following two experiments on xterm. In the first experiment, we ran an `xterm` terminal application, entered a string into the xterm, and performed the checkpointing. The string appeared in the checkpoint file 6 times. After clearing the memory of `xterm` and its child process `bash`, the string appeared in the checkpoint file 3 times.

[b] Our implementation was done on a guest VM running Ubuntu 9.10 Linux kernel version 2.6.31. In Ubuntu Linux kernel version 3.1.1, `read_buf` and `echo_buf` are removed from `tty_struct` and are stored in the structure of type `n_tty_data`.



(a)



(b)

Figure 6 Experimental results of SPARC and VirtualBox's default checkpointing mechanism.

After clearing `xterm`, `bash`, and the associated TTY buffers, the string no longer appeared in the file.

In the second experiment, we used `xterm` to run the "su" program in a VM, entered the password, and checkpointed the VM. The string appeared twice in the checkpoint file. Clearing `xterm`, `bash`, and `su` processes had no effect on the number of appearances of the string. The string disappeared after we cleared the TTY buffers.

3.1.3 Experimental Results

This section presents the results of experiments done to evaluate the performance of *SPARC* on a number of applications that may process users' confidential data. Such applications include FireFox web browser 3.5.3, ThunderBird email client 2.0.0.24, Evince document viewer 2.28.2, Gedit text editor 2.28, OpenOffice Writer word processor 3.1.1, Skype VOIP application 2.1.0.81, Gnote desktop notes software 0.5.2, and xterm terminal emulator 2.43. All experiments were conducted on a host system with Intel Dual CPU 2.26GHz processor and 2GB of RAM, and running Ubuntu Linux 10.04 kernel version 2.6.32, and a guest VM with 800MB of memory, a single processor, and Ubuntu Linux 9.10 kernel version 2.6.31.

Note that the time it takes for VirtualBox to perform checkpointing depends on the number of memory pages that are dirty; the more the number of dirty pages, the longer the checkpointing time. In our experiments, we first ran a program which allocates large amounts of memory and fills the memory with random data. We then started the application that we would like to exclude and performed checkpointing. The size of the checkpoint file was around 630 MB on average.

Figure 6 compares the overall execution time of performing checkpointing using VirtualBox's default mechanism and using SPARC, respectively. The execution time does not include the time spent in scrubbing deallocated pages of the excluded process. We have conducted a separate experiment to obtain the time spent in scrubbing pages, as demonstrated later in this section. Each data point reported is an average of execution time over 5 runs. To prevent one run from affecting the performance of subsequent runs, we deleted the previous checkpoint and rebooted the VM before we started a new run. Our experimental results show that, *SPARC* imposes $0.5\% - 7.0\%$ overhead on checkpointing, $1.4\% - 2.5\%$ overhead on restoration, and $1\% - 5.3\%$ of overall overhead.

Time spent in scrubbing memory pages: We have conducted an experiment to obtain the time spent in scrubbing 1000 pages. In our experiment, we allocated 1000 memory pages using kmalloc, filled the pages with 1's, and then scrubbed the pages. In order to get the precise execution time, we used a nanosecond precision kernel timer module and ran the experiment 10 times. The experimental results show that it takes $2.78 * 10^{(-4)}$ seconds to scrub 1000 pages on the average, which means that it takes an average of 278 nanoseconds ($2.78 * 10^{(-7)}$ seconds) to scrub one page.

3.2 Accounting for Inter-Process Communication

Processes often communicate with each other, directly or indirectly, through mechanisms such as sockets, pipes, FIFO buffers, shared memory, files, etc. Privacy-aware checkpointing must account for such inter-process communication (IPC) for two reasons, namely to prevent the leakage of confidential information and to ensure system stability after restoration. An excluded process may communicate with other processes (called *peer* processes) before checkpointing or after restoration. Prior to checkpointing, the excluded process may have communicated confidential data to peer processes. Thus, these peer processes must also be identified and excluded from the checkpoint. After restoration, a peer process may attempt to communicate with the excluded process, even though the latter does not exist in the restored VM image (by definition). In addition, after restoration, a peer process may also attempt to access files or shared memory regions that were modified by the excluded process before checkpointing. Such files could have been rolled back to a previous 'safe' state by SPARC (See Section 4), whereas such shared memory regions could have been cleared after restoration. The above situations have the potential to affect the system stability after restoration. This section describes how SPARC identifies and excludes peer processes prior to checkpointing and after restoration.

3.2.1 Detecting Inter-Process Communication Prior to Checkpointing

Prior to checkpointing, our goal is to prevent the leakage of confidential data via inter-process communication with the excluded process. To identify all peer processes prior to checkpointing, SPARC monitors system calls using kernel's `jprobes` mechanism and analyzes this information to derive inter-process dependencies. The identified peer processes will be excluded from being checkpointed if they are not 'system' processes and have not already terminated before checkpointing. System peer processes are excluded only if excluding such processes does not affect the stability of the system. A terminal shell (such as `bash`) is an example of system processes that can be excluded. If a peer process terminates before checkpointing, its memory footprint is cleared (zeroed out) using the techniques described in Section 3.

To detect communication through *files, pipes, or FIFOs*, we use jprobe to intercept `read()`, `readv()`, `write()`, and `writev()` system calls. We then record the inodes associated with file descriptors, which are passed as parameters to these system calls. If the excluded process performs a write operation on any of these IPC objects and another process reads the object whose inode has been recorded, then the latter process is tagged as a peer process since it may have received confidential data from the excluded process.

Communication through *sockets* is detected by intercepting system calls used for establishing connections and for sending or receiving data from a socket. Such system calls include `connect()`, `write()`, `writev()`, `send()`, `sendto()`, `sendmsg()`, `sendfile()`, `read()`, `readv()`, `recv()`, `recvfrom()`, and `recvmsg()`. We then examine the port numbers and IP addresses associated with sockets to determine peer processes that communicate with the excluded process.

Processes may also communicate through *shared memory*. A process can use `shmget()` system call to allocate a region of shared memory and another process can then use `shmat()` system call to map the region into its virtual memory space. The mapped region can then be read or written without using system calls (just like a process' local memory). We detect communication through shared memory by intercepting `shmget()` and `shmat()` system calls and examining (key, id) associated with each shared memory region, in which key is the the key derived from the user specified file-path, and id is generated by the system. We then mark the shared memory regions mapped by the excluded process based on (key, id). If a process maps the marked shared memory regions, then it is tagged as a peer process. The pages corresponding to the shared memory region are cleared in the checkpoint file (but not in the DRAM) and the corresponding peer processes are also excluded from the checkpoint. Inter-process communication through *semaphores* and *message queues* is similarly handled. Communication through semaphore is detected by monitoring the `semget()` and `semop()` system calls and communication through message queues is detected by monitoring `msgget()`, `msgsnd()` and `msgrcv()` system calls. Dependencies through file locks are handled in a manner similar to semaphores, except that we monitor `flock()` and `fcntl()` system calls.

In addition, we detect both direct and indirect communications. For example, assume that process P_1 writes to an IPC object O_1, process P_2 reads O_1 and writes the content of O_1 to O_2, and process P_3 reads O_2. Here P_1 and P_2 communicate with each other directly, and P_1 and P_3 communicate with each other indirectly. Thus, if P_1 is excluded from the checkpoint, P_3 is also tagged as a peer process of P_1 and is excluded from the VM checkpoint.

3.2.2 Detecting Inter-Process Communication After Restoration

Some IPC objects are persistent across process lifetimes, i.e. they allow communication between processes even if the communicating processes do not exist and execute concurrently. For example, shared memory objects persist across a process' lifetime until they are explicitly destroyed by a process. Thus an excluded process may access a shared memory object before checkpointing and a peer process may access the same object after restoration. The peer process may execute incorrectly because we clear the shared memory objects accessed by the excluded process in the checkpoint file. Similarly, the excluded process can write to a file before checkpointing and a peer process may access the file after restoration. The peer process may execute incorrectly if SPARC had restored the file to a known 'safe' state when checkpointing the disk image (Section 4).

In this section, we describe how SPARC identifies peer processes which access IPC objects that persist across the excluded process' lifetime. Our basic approach is to record all processes that exist prior to checkpointing and then terminate such processes if they attempt to communicate with the excluded process after restoration. We now describe how SPARC handles specific IPC objects after VM restoration.

- *Files and FIFOs:* When performing disk checkpointing, SPARC may exclude all files that contain confidential data from the excluded process using the technique described in Section 4. To detect IPC via files after restoration, we record the absolute paths of all files written by the excluded process prior to checkpointing. After restoration, if a process calls the `open()` system call with one of the recorded file paths as a target and without `O_CREAT` flag set (i.e. create the file if it does not exist), then the process is treated as a peer process and is terminated. Otherwise, if the `O_CREAT` flag is set, then the corresponding file path is removed from the set of recorded file paths. If a process that does not exist prior to checkpointing calls `open()` after restoration to read a file whose path was recorded, then `open` returns with an error indicating that the file cannot be accessed. FIFOs are handled similarly except that we deallocate FIFOs just before restoration. The path of the FIFO is removed from the list of the recorded paths if a process specifies the respective path as parameter to the `mkfifo()` system call used for creating FIFOs.

- *Semaphores, message queues, and shared memory:* If the excluded process calls `shmget()`, `semget()`, or `msgget()`, then we record the unique (key, id) associated with the semaphore, message queue, or shared memory segment, respectively. During restoration, we remove all semaphores, message queues, and shared memory regions accessed by the excluded process. After restoration, if a process makes any of the aforementioned system calls with the recorded (key, id) as a parameter and without `IPC_CREAT` flag set (which tells the system call to create object if it does not exist), then the process is terminated. Otherwise, if the `IPC_CREAT` flag is set, then the specified (key, id) is removed from the set of the recorded keys.

3.2.3 Experiments and Performance Results

Table 1 reports processes detected by our program, which have communicated with a number of applications, when specific actions were performed on applications. The applications include FireFox web browser 3.5.3, ThunderBird email client 2.0.0.24, Evince document viewer 2.28.2, Gedit text editor 2.28, Vim editor 7.2, Skype VOIP application 2.1.0.81, and xterm terminal emulator 2.28.1. Column "Action" gives the specific actions performed on the applications and column "Communicating processes" gives the processes that have communicated with the applications. None of the processes in the "Communicating process" column will affect the stability of the system and hence can be safely excluded from the checkpoint. All experiments were conducted on a host system with Intel Quadcore CPU 2GHz processor and 8GB of RAM, and running Windows 7, and a guest VirtualBox VM with 1024MB of memory and Ubuntu Linux 9.10.

To evaluate the overhead of our program, we measure the average time spent on intercepting system calls using jprobe and detecting processes that have communicated with the excluded process.

Excluded app	Action	Communicating processes
Gedit	Edit a file f with Gedit Save f with Gedit and then edit f with vim	gnome-panel, vim, bash
Firefox	Send an email in Gmail with Firefox	gconfd-2, Firefox children
Evince	Open a pdf file	gnome-panel
Thunderbird	Send an email	Thunderbird children
OpenOffice	Create a new file with OpenOffice Writer, save it, and then open it again	soffice.bin, Xorg
Gnome-terminal	Perform a few basic terminal operations such as mkdir, cd, touch, ls cat, cd	gnome-pty-helper, bash, cat
Skype	Send a text message and make a call	pulseaudio, gconf-helper

Table 1 Sample processes that have communicated with the excluded application.

Communication through files: To compute the average time spent in intercepting the read and write system calls using jprobe, we wrote a script which copies a file 100 times, and computed the difference between the execution time of the script without and with inter-process communication detection. We then divided the difference by 100, which is the time spent in intercepting one read and one write system call. Our experimental results show that intercepting read and write system calls does not impose an observable overhead to the system.

To evaluate the overhead of detecting write-read dependencies through files, we wrote a script in which the excluded process reads a file and writes the content of the file to a another file. The overhead of detecting one write-read dependency is 0.25 milliseconds on average.

Communication through sockets: In this experiment, we wrote a script in which a TCP client connects to a TCP server 100 times. The overhead of intercepting one connect()/bind() system call, and detecting one process dependency is 0.52 milliseconds on average.

Communication through shared memory: In this experiment, the excluded process creates a shared memory and 100 processes access the shared memory. The overhead of intercepting one shmget() system call and detecting one process dependency is 0.01 milliseconds on average.

4 Privacy-Aware Virtual Machine Disk Checkpointing

Virtual disk images of VMs may also be checkpointed along with the VM's memory image. For example, a programmer in a software development team may want to checkpoint the disk image of his/her VM and send it to his/her team member to facilitate the software development. There are several approaches for checkpointing disk state: (1) The VM freezes the current disk and creates a new differencing disk to which all subsequent write operations are redirected; (2) The VM clones the entire disk image upon every checkpointing; (3) The VM clones the disk when checkpointing is first performed, and saves changes to the disk incrementally in later checkpointing. The first approach is fastest and uses least amount of disk space to store checkpoints. However, excluding the confidential data during checkpointing results in the deletion of such data on the disk. VirtualBox checkpointing uses the first approach and its cloning mechanism uses the second approach. Amazon EC2 uses the third approach to perform volume snapshot.

Virtual disk may also contain sensitive data that should ideally be excluded from checkpoints. Some cloud computing systems such as Amazon EC2 enable users to share a disk snapshot, which increases the risk of leaking users' private data on the disk. For example, Balduzzi et al. (2012) analyzed 5303 Amazon public EC2 snapshots and found that *many of them contain confidential data such as private keys, passwords, browser history, and deleted files*. After being notified by the authors, Amazon published a tutorial on how to create and share public images in a secure manner. However, this tutorial provides only instructions on how to manually delete specific data, such as ssh keys, from disk checkpoints.

Manually excluding every single piece of confidential data from disk images/checkpoints is neither practical nor scalable. To address this issue, we propose an *application-level approach* and a *file-level approach* to prevent users' confidential data from being stored in disk checkpoints. The application-level approach enables users to exclude applications that process confidential data and all files written by such applications from disk checkpoints. Alternatively, during checkpointing, one could also replace all the files modified by such applications with files that were created during installation. This approach is independent of the format of the data (for example, whether or not the data is stored in clear-text) and does not require users to have thorough knowledge about what data the application may write to the disk. The file-level approach enables users to set aside specific directories in the virtual disk's file system to store user's confidential data as well as confidential data obtained from other users; such directories will not be checkpointed. Stability and correctness could be affected if non-sensitive applications depend on the data written to the virtual disk by a sensitive applications.

Since Virtualbox's checkpointing mechanism does not create a new copy of the disk, our implementation is based on Virtualbox's cloning mechanism. Implementing the file-level approach is straightforward and hence is not demonstrated in this section. To exclude all the files written by an application from disk checkpoints, we have developed a user-space daemon process to record files modified and created during the execution of an application as follows. When the daemon process detects that there is an *apt-get* process running, it forks an *strace* process and attaches it to the *apt-get* process to monitor all files created by *apt-get* and its child processes. When a user performs cloning, we first use VirtualBox's cloning mechanism to create a raw disk image of the VM. Next, we mount the raw disk image and use the *shred* program to permanently delete files modified and created during the execution of the application. Finally, the user can either create a new VM based on the raw disk image or attach the raw disk image to the original VM. Replacing all files modified by an application with files that were created during the installation of the application is handled similarly, except that (1) in addition to recording a set $fset_1$ of all files modified and created during the execution of the application, we also record a set $fset_2$ of all files created during the installation of an application, and (2) after mounting the raw disk image, we replace files in $fset_2$ with corresponding files in $fset_1$ and delete files that are in $fset_2 \setminus fset_1$.

5 Related Work

Previous work on minimizing data lifetime (e.g.,Garfinkel et al. (2004); Chow et al. (2005); Solomon and Russinovich (2000); Chow et al. (2004); Tang et al. (2012)) has focused on clearing the deallocated memory (also known as memory scrubbing). However, memory scrubbing does not solve the problem of confidential data being checkpointed *before* the pages are deallocated. As a result, *SPARC* also clears memory pages of the excluded process

in checkpoints. Selectively clearing memory pages during checkpointing is much more challenging than scrubbing only deallocated memory because multiple processes may share the same memory pages and we must ensure that excluding one process will not affect other processes when the VM is restored.

Hu et al. (2013) presented an application-level privacy-preserving virtual machine checkpointing mechanism, which allows applications to control the granularity at which their confidential data is excluded from VM checkpoints. This approach, however, is not application-transparent. Davidoff (2008) showed that RAM may retain confidential data even after the system has been powered off. This permits cold boot memory dumping attacks, assuming that the attacker has physical access to the machine during the limited time when the RAM retains information without power. The problem is more severe with VM checkpoints because the memory snapshot is saved to a persistent storage, potentially exposing confidential data forever, and an attacker does not need physical access to the machine.

Features protecting virtual disk, memory, and checkpoints have found their way into research prototypes as well as commercial virtualization products. Garfinkel et al. (2003) developed a hypervisor-based trusted computing platform, whose privacy features include encrypted disks and the use of a secure counter to protect against file system rollback attacks. Encrypting checkpoints has also been recommended in Garfinkel and Rosenblum (2005). However, encrypting the checkpoint alone is insufficient because (1) it still prolongs the lifetime of confidential data that should normally be quickly destroyed after use; (2) when the VM is restored, the checkpoint will be decrypted and loaded into the memory of the VM, thus exposing the confidential data again; (3) the checkpoint file may be shared by multiple users, thus increasing the likelihood of data leakage. Garfinkel et al. (2004) also proposes to encrypt confidential data in the memory and clear such data by discarding the key. However, encrypting data in memory can add significant overheads to access the information and may still expose sensitive information since VM checkpointing can occur just when a program decrypts the data. Gallagher (1992) outlined security requirements for reusing deallocated memory pages without risk of exposing confidential data that may linger in memory. Our approach goes beyond it by also considering sensitive memory that has not been deallocated.

A large body of literature considers checkpointing and replaying the execution of processes, as means for intrusion detection, debugging, process migration, and fault tolerance (e.g. Bozyigit and Wasiq (2001); Dunlap et al. (2002); King et al. (2005); Laadan et al. (2010); Osman et al. (2002); Bressoud and Schneider (1995); LeBlanc and Mellor-Crummey (1987); Xu et al. (2007)). However, none of them examine the data lifetime implications of checkpointing.

Chen et al. (2008) proposed a mechanism called *Overshadow* to protect the memory of applications from the operating system, by encrypting the memory of applications when switching to the system context. Our approach is different from theirs by focusing on eliminating confidential data in both user-level applications and the kernel. Ristenpart and Yilek (2010) presented a VM reset attack in which a server may send the same randomness (e.g. session key) to two different web servers when the victim VM is restored twice from the same checkpoint file. Kangarlou et al. (2012) presented techniques for taking snapshots of virtual networked infrastructures in the cloud. However, they did not address the privacy issue of checkpoints.

Our approach for detecting inter-process communication differs from the approach in Zheng and Nieh (2004) as follows: (1) we monitor system calls using the kernel's

`jprobes` mechanism, while they manually override the kernel's function pointer to a system call; (2) we consider inter-process communication through shared memory, while they did not; (3) we develop mechanisms for detecting inter-process communication after VM restoration; their approach cannot be directly applied to address this issue.

This paper extends Gofman et al. (2011) in several ways: (1) we have developed, implemented, and evaluated techniques for tracking process dependencies due to inter-process communication and accounted for such dependencies during VM checkpointing and restoration operations; (2) we developed a privacy-aware VM disk checkpointing mechanism; (3) we conducted experimental evaluation of the mechanism for scrubbing memory pages; and (4) we presented a threat model and used a real-world scenario to illustrate the vulnerability of VM checkpointing.

6 Conclusions and Future Work

This paper presents *SPARC*, a security and privacy aware VM checkpointing mechanism, which enables users to selectively exclude processes, terminal applications, and disk contents that contain users' confidential data from being checkpointed. We have implemented a prototype of *SPARC* on the VirtualBox hypervisor and Linux VM and tested it over a number of applications. Our preliminary results show that *SPARC* imposes only $1\% - 5.3\%$ of overhead with common application workloads.

The techniques presented in the paper are useful when the applications within the VMs cannot be modified. Since the semantics of the application internals are unknown, this approach requires that the application be terminated when the VM is later restored, because the integrity of the application cannot be guaranteed upon resumption from a sanitized checkpoint. However, in some situations, it may be desirable to exclude only memory locations storing confidential data, while keeping the application alive after the VM is restored. We plan to develop techniques to address this issue. The challenges are how to efficiently identify all memory locations and disk contents storing confidential data and how to clear the data without crashing the process after restoration. We also plan to design a light-weight process container that cleanly encapsulates the state of each process (or process groups), which would help avoid scrubbing process-specific information from disparate locations in OS memory. Finally, we will identify potential attacks that may specifically target *SPARC* to hide the attacker's activities and develop counter-measures.

Acknowledgment This work is supported in part by the National Science Foundation through grants CNS-0845832, CNS-0855204, and CNS-1320689.

References

M. Balduzzi, J. Zaddach, D. Balzarotti, E. Kirda, and S. Loureiro. A security analysis of amazon's elastic compute cloud service. In *ACM Symposium on Applied Computing*, pages 1427–1434, 2012.

M. Bozyigit and M. Wasiq. User-level process checkpoint and restore for migration. *SIGOPS Oper. Syst. Rev.*, 35(2):86–96, 2001.

T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *The fifteenth ACM symposium on Operating systems principles*, SOSP '95, pages 1–11, 1995.

Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R. K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of USENIX Security Symposium*, pages 22–22, 2004.

Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Shredding your garbage: reducing data lifetime through secure deallocation. In *Proceedings of the USENIX Security Symposium*, pages 22–22, 2005.

Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005. ISBN 0596005903.

Sherri Davidoff. Cleartext passwords in linux memory. http://www.philosecurity.org, 2008.

Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *15th ACM conference on Computer and communications security*, pages 51–62, 2008. ISBN 978-1-59593-810-7.

George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *In Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, pages 211–224, 2002.

Patrick R. Gallagher. A guide to understanding object reuse in trusted systems. 1992.

Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium, pages*, pages 191 – 206, 2003.

Tal Garfinkel and Mendel Rosenblum. When virtual is harder than real: security challenges in virtual machine based computing environments. In *Proceedings of the 10th conference on Hot Topics in Operating Systems*, pages 20–20, 2005.

Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. pages 193–206. ACM Press, 2003.

Tal Garfinkel, Ben Pfaff, Jim Chow, and Mendel Rosenblum. Data lifetime is a systems problem. In *Proc. of ACM SIGOPS European workshop*. ACM, 2004.

Mikhail I. Gofman, Ruiqi Luo, Ping Yang, and Kartik Gopalan. Sparc: a security and privacy aware virtual machinecheckpointing mechanism. In *Proceedings of the 10th annual ACM workshop on Privacy in the electronic society*, pages 115–124, 2011.

Yaohui Hu, Tianlin Li, Ping Yang, and Kartik Gopalan. An application-level approach for privacy-preserving virtual machine checkpointing. In *the 6th IEEE International Conference on Cloud Computing*, pages 59 – 66, 2013.

Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 91–104, 2005.

Ardalan Kangarlou, Patrick Eugster, and Dongyan Xu. Vnsnap: Taking snapshots of virtual networked infrastructures in the cloud. In *IEEE Transactions on Services Computing*, volume 5, pages 484–496, 2012.

Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 1–15, 2005.

Kenichi Kourai and Shigeru Chiba. Hyperspector: Virtual distributed monitoring environments for secure intrusion detection. In *ACM/USENIX International Conference on Virtual Execution Environments*, pages 197 – 207, 2005.

Oren Laadan, Nicolas Viennot, and Jason Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *the ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 155–166, 2010.

T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, 36:471–482, April 1987.

Anh M. Nguyen, Nabil Schear, HeeDong Jung, Apeksha Godiyal, Samuel T. King, and Hai D. Nguyen. Mavmm: Lightweight and purpose built vmm for malware analysis. In *Annual Computer Security Applications Conference*, pages 441–450, 2009.

Daniela Alvim Seabra de Oliveira and S. Felix Wu. Protecting kernel code and data with a virtualization-aware collaborative operating system. In *Annual Computer Security Applications Conference*, pages 451–460, 2009. ISBN 978-0-7695-3919-5.

Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The design and implementation of zap: A system for migrating computing environments. In *Symposium on Operating Systems Design and Implementation (OSDI 2002)*, pages 361–376, 2002.

Bryan D. Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. Lares: An architecture for secure active monitoring using virtualization. In *IEEE Symposium on Security and Privacy*, pages 233 – 247, 2008.

Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *International symposium on Recent Advances in Intrusion Detection*, pages 1–20, 2008.

Thomas Ristenpart and Scott Yilek. When good randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2010.

Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proceedings of Twenty-First ACM SIGOPS symposium on Operating Systems Principles*, pages 335–350, 2007.

David A. Solomon and Mark Russinovich. *Inside Microsoft Windows 2000*. Microsoft Press, 2000. ISBN 0735610215.

Yang Tang, Phillip Ames, Sravan Bhamidipati, Ashish Bijlani, Roxana Geambasu, and Nikhil Sarda. Cleanos: limiting mobile data exposure with idle eviction. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 77–91, 2012.

Min Xu, Vyacheslav Malyugin, Jeffrey Sheldon, Ganesh Venkitachalam, Boris Weissman, and Vmware Inc. Retrace: Collecting execution trace with virtual machine deterministic replay. In *In Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation, MoBS*, 2007.

Haoqiang Zheng and Jason Nieh. Swap: a scheduler with automatic process dependency detection. In *Symposium on Networked Systems Design and Implementation*, pages 14–14, 2004.