SPARC: A Security and Privacy Aware Virtual Machine Checkpointing Mechanism

Mikhail I. Gofman
mgofman1@binghamton.edu

Ruiqi Luo
rluo1@binghamton.edu

Ping Yang
pyang@binghamton.edu

Kartik Gopalan
kartik@binghamton.edu

Department of Computer Science, State University of New York at Binghamton, NY 13902, USA

ABSTRACT

Virtual Machine (VM) checkpointing enables a user to capture a snapshot of a running VM on persistent storage. VM checkpoints can be used to roll back the VM to a previous "good" state in order to recover from a VM crash or to undo a previous VM activity. Although VM checkpointing eases systems administration and improves usability, it can also increase the risks of exposing sensitive information. This is because the checkpoint may store VM's physical memory pages that contain confidential information such as clear text passwords, credit card numbers, patients' health records, tax returns, etc.

This paper presents the design and implementation of SPARC, a security and privacy aware checkpointing mechanism. *SPARC* enables users to selectively exclude processes and terminal applications that contain sensitive data from being checkpointed. Selective exclusion is performed by the hypervisor by sanitizing memory pages in the checkpoint file that belong to the excluded applications. We describe the design challenges in effectively tracking and excluding process-specific memory contents from the checkpoint file in a VM running the commodity Linux operating system. Our preliminary results show that *SPARC* imposes only $1\% - 5.3\%$ of overhead if most pages are dirty before checkpointing is performed.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection;
K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Security

Keywords

Security, Privacy, Virtual Machine Checkpointing

WPES'11, October 17, 2011, Chicago, Illinois, USA.

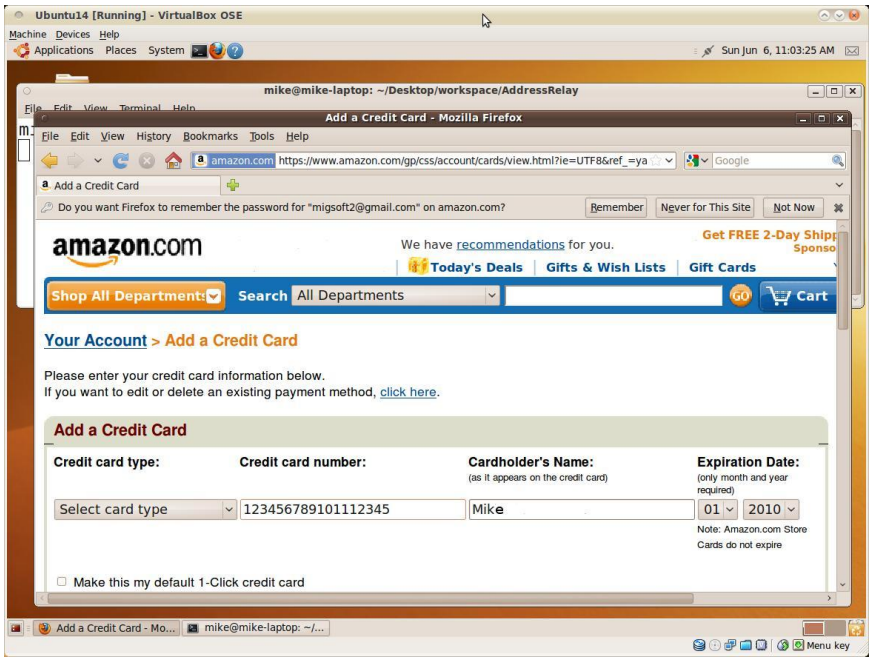
1. INTRODUCTION

Virtualization technology is being widely adopted in grid and cloud computing platforms [42, 46, 31, 38] to improve server consolidation and reduce operating costs. On one hand, virtual machines (VMs) help improve security through greater isolation and more transparent malware analysis and intrusion detection [30, 32, 37, 14, 15, 19, 22, 39, 35, 24]. On the other hand, virtualization also gives rise to new challenges in maintaining security and privacy in virtualized environments. 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.

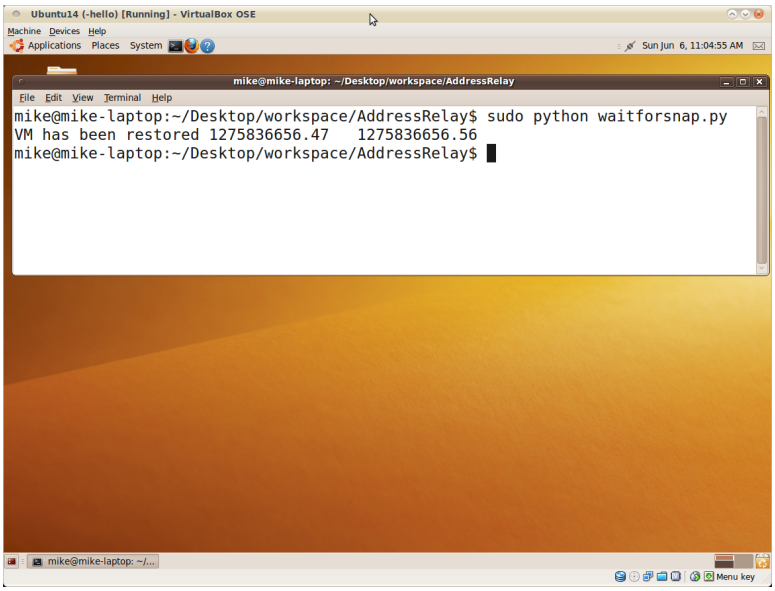
VM checkpointing saves a persistent snapshot (or a checkpoint) of the entire memory and disk state of a VM in execution. VM checkpointing enables a user to recover a long-running process after the process crashes and to roll back to an earlier "good" state if the user wishes to undo prior operations or patches on the VM. Most hypervisors such as VMware [43], Hyper-V [29], VirtualBox [33], KVM [1], and Xen [3] support VM checkpointing.

Despite its many benefits, VM checkpointing also has its drawbacks from security viewpoint. Checkpoints are stored on persistent storage and contain the VM's physical memory contents at a given time instant. Hence persistent checkpoints can drastically prolong the lifetime of memory pages containing sensitive information such as clear text passwords, credit card numbers, and other sensitive information which would normally be quickly discarded after usage. For example, in our experiments, we studied VirtualBox's memory checkpoints with a hex editor, and found plain text credit card numbers in the memory of Firefox web browser and passwords in the memory of `xterm` terminal emulator (both running and terminated). In addition, sensitive information may also linger in pages of terminated processes because pages were not deallocated or page data was not cleared after deallocation. Prior research on minimizing the data lifetime has primarily focused on clearing the deallocated memory [17, 9]. However, this does not prevent memory pages from being checkpointed before they are deallocated. Some other work [20, 2] proposed to protect the checkpointed information by encrypting the checkpoint files. However, when the VM is restored, the checkpoint file will be decrypted and loaded into the memory of the VM, thus making the sensitive information vulnerable again. Worse still, if an attacker hacks the user's account, the attacker will be able to obtain all the information being checkpointed by simply restoring all checkpoints.

In this paper, we present *SPARC*, a Security and Privacy Aware



(a)



(b)

Figure 1: VM restored using (a) VirtualBox's default checkpointing mechanism; and (b) *SPARC* with Firefox excluded.

Checkpointing mechanism, which enables users to exclude specific applications that contain users' confidential information from being checkpointed by the hypervisor. For example, a user may wish to exclude a web browser application from being checkpointed because the user may enter his or her password or credit card number. Moreover, *SPARC* enables users to exclude terminal applications on which applications processing sensitive information are running from being checkpointed. We have implemented a SPARC prototype based on the VirtualBox 3.1.2_OSE hypervisor and Ubuntu Linux 9.10 guest (kernel v2.6.31). The experimental results show that *SPARC* imposes only $1\% - 5.3\%$ checkpointing overhead with common application workloads.

Organization.

The rest of the paper is organized as follows. Section 2 and Section 3 present techniques for excluding user applications and terminal processes from being checkpointed, respectively. Section 4 presents our experimental results with *SPARC*. Section 5 gives the related work and the concluding remarks appear in Section 6.

2. EXCLUDING AN APPLICATION FROM BEING CHECKPOINTED

SPARC enables users to specify applications they wish to exclude from being checkpointed. Such applications are typically applica-

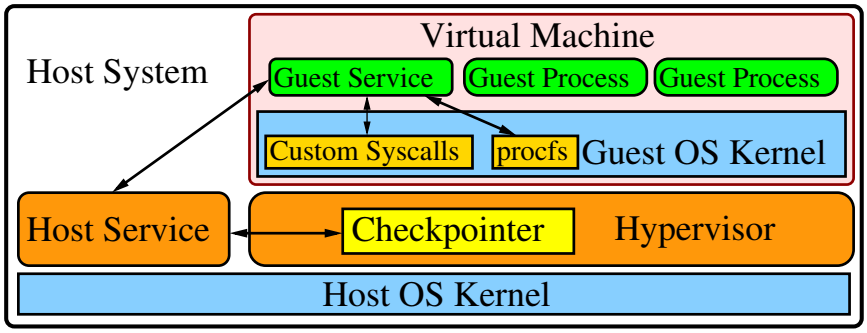


Figure 2: The Architecture of *SPARC*

tions that may process sensitive information (e.g. Firefox, Internet Explorer, Email clients, etc). VirtualBox checkpointing creates two files: a `.sav` file which stores the contents of the VM's physical memory, and a `.vdi` file which stores the disk image. For efficiency, when checkpointing the disk image, instead of cloning the entire disk, VirtualBox freezes the current disk and creates a new *differencing disk* to which all subsequent write operations are redirected. In this paper, we focus on excluding physical memory of specific applications from being checkpointed and leave disk checkpointing issues for future work.

Consider an example where a user has entered a credit card number into the Firefox web browser. If the user performs checkpointing after the credit card number is entered, then the credit card number may be stored in the checkpoint even if Firefox has been terminated or is being used to access other URLs. *SPARC* would enable the user to exclude Firefox from being checkpointed by not storing the data processed by Firefox in the checkpoint file. In addition, *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 1(a) gives the screenshot of a VM restored using VirtualBox's default mechanism, in which checkpointing is performed as soon as the user enters his or her credit card number. Figure 1(b) gives the screenshot of the VM restored using *SPARC* in which Firefox and the information processed by Firefox are excluded from being checkpointed.

Figure 2 gives the high-level architecture of *SPARC*. First, the user selects a list of applications that he or she wishes to exclude from being checkpointed. Next, a special process called the *guest service* in the VM invokes custom system calls[1] to identify and collect physical addresses of memory pages that belong to the application being excluded, such as process memory, page cache pages, etc. Checkpointing is initiated from another special process called the *host service* located at the host system. The host service sends a notification to the guest service that checkpointing has been requested. The guest service replies with the collected physical addresses of memory pages that need to be excluded. The host service then relays the addresses to the hypervisor which in turn commences the checkpoint. The checkpointer in the hypervisor uses the received physical addresses to determine which memory to clear in the `.sav` file. To ensure that the VM can be restored successfully, excluding a process should not affect other processes that do not communicate with the excluded process. As a result, memory pages that are shared by multiple processes will not be excluded from being checkpointed.

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

Excluding process physical memory.

Figure 3 shows the virtual address space layout of a process in Linux. The *program code* segment stores the executable code of the program. The *uninitialized* and *initialized* data sections store uninitialized and initialized static variables, respectively. The *heap* contains dynamically allocated memory. The *memory mapped files* section contains files mapped into the memory of the process (e.g. shared libraries). The *stack* contains information about function calls in progress e.g. local variables of functions.

Below, we describe how *SPARC* identifies and collects information about memory pages that belong to a process with ID `pid` and excludes those pages from the checkpoint file. First, the guest service invokes a system call that locates the process descriptor (i.e. `struct task_struct`) associated with the process, which links together all information of a process e.g. memory, open files, associated terminal, and pending signals. From the process descriptor, we obtain a memory descriptor (`struct mm_struct`) which contains the starting and ending addresses of each segment of process virtual memory.

Next, the guest service breaks up the memory of each segment into its constituent virtual pages and obtains the physical address of each page. We skip over the segment representing the executable image because it does not contain sensitive information and clearing it may affect other processes that share the same in-memory executable image. We also skip over the segments representing memory-mapped files (usually libraries) because clearing these segments may affect other processes that map the same files into their memory. The guest service 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. For each virtual page, the file contains information about whether the page is resident in the physical memory and, if so, provides the physical address of the page. To avoid affecting other processes in the system, we skip all resident pages which are being mapped more than once. To determine the number of times a physical page has been mapped, the guest service checks the file `/proc/kpagecount` which records the number of times each physical page has been mapped.

Finally, the physical address of each page is sent to the host service which in turn relays the address to the hypervisor. When VirtualBox creates a memory checkpoint, prior to saving a physical page to the `.sav` file, *SPARC* checks if the physical address of the page

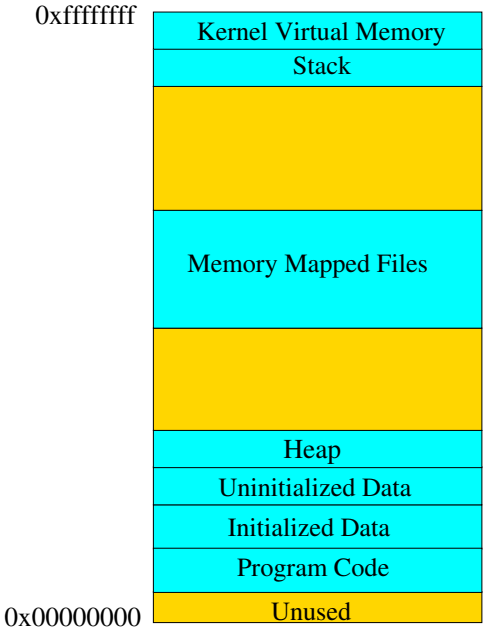


Figure 3: Process virtual memory layout

matches one of the received addresses. If not, it saves the contents of the page to the checkpoint file. Otherwise, it saves a page containing all 0's. To implement this behavior in the VirtualBox, we have modified the function `pgmSavePages()`. Because pages are constantly swapped between the disk and the physical memory, the virtual-to-physical memory mappings of a process may change after collecting the physical addresses. This may result in excluding the wrong memory contents. We overcome this by freezing all user space processes except the guest service prior to gathering the physical addresses. This is achieved by using the *freezer subsystem* [45] of the kernel, designed for selectively freezing user-level processes for hibernation or resource management purposes. Once the checkpointing completes, all processes are unfrozen and the execution proceeds as normal.

When the VM is restored, the guest service detects the restoration event and sends the `SIGKILL` signal to each process whose memory contents were previously excluded during checkpointing. This `SIGKILL` signal is required to allow the guest kernel to clean up any residual state (other than memory) for excluded processes before the VM resumes. Finally, the guest service unfreezes the all processes and the execution proceeds as normal. Note that, if the process being excluded deallocates pages containing sensitive information prior to the checkpointing, these pages can no longer be identified and cleared. To solve this problem, we modify kernel functions to zero out deallocated pages belonging to the process being excluded 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. Page cache speeds up disk operations as follows. When data is read from the disk, a page is allocated in the physical memory and is filled with the corresponding data from the disk. Thus, all subsequent read operations targeted at the same disk location can quickly access the data in the main memory. Subsequent write operations to the disk location simply modify the page in the page cache. The kernel, after some delay, synchronizes the page with the disk. Every disk operation in Linux goes through the page cache (except for the swap device or files opened with `O_DIRECT` flag) [4].

If the process performs disk I/O operations, the sensitive information read from and written to the disk 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 retains 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. Even when the page is evicted, the page contents will remain in the free memory pool until they are overwritten.

SPARC excludes the cached pages of a process in the checkpoints as follows. First, it retrieves the file descriptor table (`struct file ** fd`) from the process descriptor of the process, which comprises an array of 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 the 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, whenever a file descriptor is closed, we evict and clear all pages from the page cache associated with the closed file descriptor. In addition, even after a page is being evicted from page cache (using `remove_from_page_cache()`), the physical memory pages may still retain sensitive data belonging to the process. Thus, *SPARC* sanitizes (zeros out) each evicted page that was originally brought into the cache on behalf of the process being excluded. Finally, the (cleared) pages in the page cache may also be used by other processes. To avoid affecting the processes which rely on these pages, when the VM is restored (but before the processes are thawed), we flush all pages used by the processes to be excluded from the page cache.

Excluding pipe buffers.

Pipes and FIFOs are mechanisms commonly used for implementing producer/consumer relationship between two processes. A pipe enables communication between the parent and the child processes. A parent process creates a pipe by issuing a `pipe()` system call. The system call returns two file descriptors. Any data written to the first file descriptor (e.g. via the `write()` system call) can be read from the second descriptor (e.g. with the `read()` system call). Shell programs make use of pipes to connect output of one process to the input of another (e.g. "`ls | grep myfile`"). Firefox browser also uses pipes to trace `malloc()` memory allocations.

FIFOs are similar to pipes but allow communication of two unrelated processes. An FIFO is created via `mkfifo()` system call, which takes the name of the FIFO as one of the parameters. Once created, the FIFO appears like a regular file on the file system, but behaves like a pipe: the producer process opens the FIFO "file" for writing and the consumer process for reading. 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`. The 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 the Google Chrome to implement communications between the renderer process and the browser process [21].

Data exchanged via pipes and FIFOs flows through a *pipe buffer* in the kernel. Hence, if the process being excluded makes use of pipes and/or FIFOs, we must also sanitize the corresponding pipe buffers. The pipe buffers are sanitized as follows. First, we locate the file descriptors opened by the process that represent pipes and FIFOs, in a manner similar to identifying file descriptors representing regular (i.e. on-disk) files. We then retrieve the associated pipe buffer descriptors (of type `struct pipe_buffer`) from each file descriptor. Finally, we retrieve the descriptors of pages used to store inter-process data from each pipe buffer descriptor and convert page descriptors into the physical addresses of pages they represent.

Excluding socket buffers.

All application-level network communication takes place through network sockets. With each socket, the kernel associates a list of socket buffers (`struct sk_buffs`) which contain data exchanged over the socket. If a process sends or receives sensitive information via an open socket (e.g. through `read()` and `write()` system calls), the information may be stored in the `sk_buffs` 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 associated with `sk_buffs`.

Identifying file descriptors of a process that represent sockets is similar to detecting pipes and FIFOs. First, we retrieve the associated socket descriptor (`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`) and the queue of socket buffers that have been received via socket (`struct sk_buff_head sk_receive_queue`). We then traverse both queues and determine the physical memory address where the payload data is stored for each socket buffer.

GUI related issues.

It is common for processes to display sensitive information on the screen. When a VM is restored, but before the process is terminated, the information displayed by the process may linger on the screen for a brief moment. To address the problem, at checkpointing time, we invoke the `XCreateWindow()` API provided by X-Windows to visually cover the windows of the processes being excluded with black rectangles. When the checkpointing completes, the rectangles are removed and the user continues using the process. When the VM is restored, the windows remain covered. We remove the windows briefly after sending the `SIGKILL` signals to the processes being excluded and unfreezing the processes. To detect all windows of a given process, we traverse the list of all open windows and check the windows' `_NET_WM_PID` property - the process ID of the process owning the window.

SPARC also enables a user to choose the process to exclude from checkpointing by clicking on the process window. When the user clicks the window, *SPARC* automatically checks the `_NET_WM_PID` property of the window and the process is then excluded as previously described.

Note that the buffers belonging to the X-windows, GTK, and other GUI components may also contain sensitive information of the process encoded in a different format. Currently we only zero out pages in the checkpoints that contain clear text. Zeroing out pages that contain sensitive information with different format can use a similar approach and is deferred to future work.

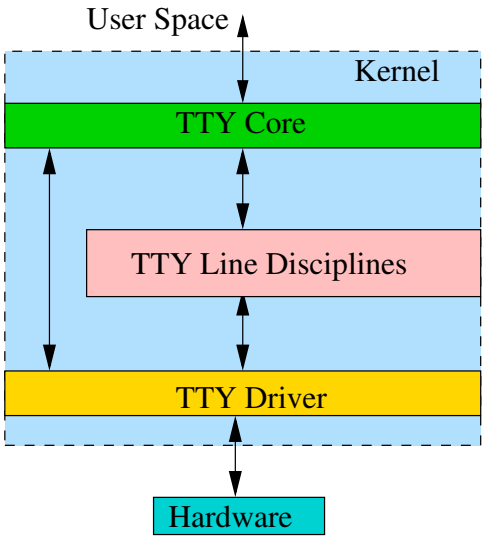


Figure 4: Teletype (TTY) subsystem architecture

3. EXCLUDING TERMINAL APPLICATIONS

Applications running on terminals may take confidential data as inputs and output confidential data on the terminal. As a result, terminals where the excluded applications are running should also be excluded from being checkpointed.

In Linux, there are two main types of terminals: *virtual consoles* and *pseudo terminals*. A system typically contains 7 virtual consoles (named `tty1-tty7`); the first 6 consoles usually provide a text terminal interface consisting of the login and shell, and the 7th console usually provides a graphical interface. Pseudo terminal applications emulate 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`), the shell forks off a child process and replaces the child's executable image with the code of the specified command. In all terminal types, by default, the child process inherits the terminal of its parent process. In this paper, we consider two of the most often used terminals: virtual consoles and terminal emulators.

All terminals rely on the Teletype (TTY) subsystem in the kernel. Figure 4 shows the architecture of the TTY subsystem where arrows indicate the flow of data. The uppermost layer of the TTY subsystem is the *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 discipline drivers*, which usually 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. There are three types of TTY drivers: console, serial port, and pseudo terminal (`pty`). 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 [11].

Identifying Terminal where a Process is Running.

The kernel associates a TTY structure (`tty_struct`) with each process descriptor, 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. If the process is running on the virtual console, then the name is "ttyxx" where "xx" is a number. If the process is running on a pseudo terminal, then the terminal name is "ptsxx".

Once we determine the terminal name where the process that needs to be excluded is running, we identify all other processes that are running on the same terminal and exclude them from being checkpointed. We achieve this by traversing all process descriptors and comparing the name of their terminals to that of the process being excluded. If the process is running on a pseudo terminal, we also exclude the pseudo terminal application (e.g. `xterm`) because it may contain the input or output information of the process. The terminal application is usually not attached to the same terminal as the process being excluded. However, the terminal application can be detected by following the pointer in the descriptor of the process running on the terminal (`task_struct * real_parent`), 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 2.

Excluding TTY Information.

We sanitize the TTY subsystem associated with the console/pseudo terminal by clearing the buffers used at each level of the TTY subsystem shown in Figure 4. The TTY subsystem maintains the following buffers: TTY core uses specialized buffers (of type `struct tty_buffer`) to store information received from the user space; TTY line discipline drivers use three respective character 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 the characters received from the device that need to be echoed back to the device (`echo_buff`).

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

Excluding pseudo terminals is slightly more complex than excluding virtual consoles because we must also sanitize the pseudo terminal driver a.k.a `pty`. The pseudo terminal driver is a specialized inter-process communication channel consisting of two cooperating virtual character devices: *pseudo terminal master (*`ptm`*)* and *pseudo terminal slave (*`pts`*)*. Data written to the `ptm` is readable from the `pts` and vice-versa. Therefore, in a terminal emulator, a parent process can open the `ptm` end of the `pty` 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. Both `pts` and `ptm` devices are associated with separate instances of TTY subsystems. After identifying the TTY subsystem instances of both devices, the rest of the operations are similar to operations involved in excluding a virtual console.

In addition, sensitive data may persist in the TTY subsystem buffers even after they are deallocated. Thus, to prevent such data from being checkpointed, we modify functions that deallocate such buffers to clear the buffers prior to deallocation.

Experiments.

We performed the following two experiments. In the first experiment, we ran an `xterm` terminal application, entered a string into the xterm prompt, and checkpointed the VM. The string appeared in the `.sav` file 6 times. After clearing the memory of `xterm` and its child process `bash`, the string appeared in the `.sav` file 3 times. After zeroing out `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 which is used to gain root privileges, entered the password into the `su`'s prompt, and created a checkpoint. The string appeared twice. Clearing `xterm`, `bash`, and `su` processes had no effect on the number of appearances. Once we cleared the TTY buffers, the string disappeared.

4. EXPERIMENTAL RESULTS

In this section, we evaluate the performance of *SPARC* on a number of applications that may process sensitive information: Firefox web browser, Thunderbird email client, Evince document viewer, Gedit text editor, OpenOffice Writer word processor, Skype VOIP application, Gnote desktop notes software, and xterm terminal emulator. All experiments were conducted on a host system with Intel Dual CPU 2.26GHz processor, 2GB of RAM, and 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.

Tables 1(a) and 1(b) give the execution time when performing checkpointing using VirtualBox's default mechanism and using SPARC, respectively. 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 start a new run.

Note that the time it takes for VirtualBox to perform checkpointing depends on the number of memory pages that are dirty; the more pages are dirty, the longer time the checkpointing is performed. In our experiments, prior to checkpointing, we run a program which allocates large amounts of memory and fills the memory with random data and then start the application that we want to exclude. Simultaneously, other typical Linux system processes are also running inside the VM. The average size of `.sav` files after checkpointing is around 630 MB.

The column heading "Operations" in these two tables gives the various operations performed. In particular, in Table 1(b), operations 1-12 and 13-18 are conducted by the guest and host services to perform checkpointing respectively, operations 19-21 are performed by the guest service to restore the VM. Rows 22 and 23 in Table 1(b) give the overall checkpointing time and the overall restoration time, respectively. Note that, because some of the operations are performed in parallel by the guest and the host service, the numbers in row 22 are slightly higher than the actual execution time.

Observe from Tables 1(a) and 1(b) that, *SPARC* imposes $0.5\%-7.0\%$ overhead on checkpointing, $1.4\%-2.5\%$ overhead on restoration, and $1\% - 5.3\%$ of overall overhead. The overheads of *SPARC* can be further reduced by using system-specific optimizations. For example, in VirtualBox the overhead of communication between host and guest services can likely be reduced by using the

Operations	Execution Time (second)							
	Firefox	Thunderbird	Evince	Gedit	OpenOffice	Skype	Gnote	Xterm
Checkpointing	16.13	16.38	16.91	16.65	15.76	16.59	17.18	17.40
Restoration	10.45	12.18	13.02	9.91	10.49	10.30	9.97	12.05

(a)

	Operations	Execution Time (second)							
		Firefox	TB	Evince	Gedit	OO	Skype	Gnote	Xterm
1	Receive checkpoint notification from host	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
2	Identify processes running on a terminal	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0.03
3	Freeze all user processes	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
4	Get physical page addresses of the process	0.11	0.10	0.10	0.10	0.08	0.08	0.09	0.14
5	Get page cache pages of the process	0.04	0.03	0.04	0.05	0.03	0.04	0.03	0.06
6	Get physical addresses of TTY buffers	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0.03
7	Get physical addresses of pipe buffers	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
8	Get physical addresses of socket buffers	0.02	0.03	0.02	0.02	0.02	0.02	0.02	0.03
9	Send physical address information to host	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00
10	Notify host service that all addresses were sent	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.04
11	Receive notification that snapshot is complete	0.01	0.01	0.02	0.01	0.00	0.00	0.00	0.00
12	Unfreeze processes	0.04	0.03	0.04	0.04	0.05	0.04	0.03	0.02
13	Send checkpoint notification to the guest service	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
14	Receive physical addresses from the guest service	0.35	0.30	0.34	0.32	0.29	0.32	0.30	0.40
15	Receive notification that addresses were sent	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.04
16	Create a checkpoint with the process excluded	15.96	16.20	16.48	17.25	15.76	16.08	16.61	17.02
17	Notify the guest that the checkpointing is completed	0.10	0.10	0.05	0.10	0.09	0.08	0.11	0.10
18	Receive notification that the checkpointing is completed	0.04	0.04	0.04	0.03	0.04	0.04	0.04	0.04
19	Kill the excluded process	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
20	Flush the page cache	0.11	0.11	0.10	0.07	0.14	0.12	0.07	0.08
21	Unfreeze processes	0.10	0.08	0.09	0.11	0.09	0.05	0.08	0.05
22	Checkpointing (Overall)	16.70	16.82	17.15	17.92	16.40	16.72	17.26	17.97
23	Restoration (Overall)	10.71	12.41	13.26	10.13	10.76	10.52	10.17	12.22

(b)

Table 1: (a) Execution time for performing checkpointing using VirtualBox's checkpointing mechanism. (b) Execution time for performing checkpointing using *SPARC*.

Host-Guest communication mechanism. This however, comes with cost of added implementation complexity.

5. RELATED WORK

A large body of literature considers checkpointing and replaying the execution of processes, as means for intrusion detection, debugging, process migration, and fault tolerance [5, 15, 16, 23, 26, 34, 28, 10, 25, 41, 6, 27, 13, 47, 48]. However, none of them examine the data lifetime implications of checkpointing or replaying the execution.

Chen et al. [7] proposed a mechanism called *Overshadow*, which protects the memory of applications from the operating system, by encrypting the memory of applications when switching to the system context. Our approach, on the other hand, focuses on eliminating sensitive information from the checkpoints which include data from both the user-level applications and the system.

Features protecting virtual disk, memory, and checkpoints have found their way into research prototypes as well as commercial virtualization products. Garfinkel et al. [18] developed a hypervisor-based trusted computing platform that uses trusted hardware features to permit systems with varying security requirements to execute side-by-side on the same hardware platform. The platform's privacy features include encrypted disks and the use of a secure counter to protect against file system rollback attacks in which the state of a file is rolled back. [20] and [2] also suggested encrypting checkpoints. However, encrypting checkpoints will not prevent sensitive information stored in these checkpoints from reappearing in the memory when the VM is restored. VMware ACE [2], VMware Infrastructure [44], and VirtualBox [33] allow users to exclude the entire memory from being checkpointed. However, none of them provide a level of granularity that we do by selectively excluding processes from the checkpointed memory.

Issues related to data lifetime have also been addressed in prior efforts. Chow et al. [8] and Garfinkel et al. [17] discussed in depth the problem of sensitive data being stored in memory, and observed that the sensitive data may linger in memory for extended periods

and hence may be exposed to compromise. [17] also proposed to encrypt sensitive information in the memory and clear the sensitive information by simply discarding the key. However, encrypting sensitive information in memory can add significant overheads to access the information and may still expose sensitive information if the VM is checkpointed at the moment when some program decrypts the sensitive information. Patrick et al. [36] outlined a set of security requirements for reusing deallocated memory resources without risk of exposing sensitive information that may linger in memory i.e. the *object reuse problem*. Our approach goes beyond the scope of object reuse problem by also considering sensitive memory that has not been deallocated. In [9], authors proposed a multi-level approach to clearing deallocated memory at the application, compiler, library, and system levels. A similar mechanism is included in Windows operating systems, which uses system idle time to clear deallocated memory pages [40]. Also, in Unix systems, it is common to clear memory before reuse [17]. However, simply clearing deallocated memory does not solve our problem because memory pages that have not been deallocated may contain sensitive information and such information may be checkpointed. As a result, *SPARC* also clears the memory pages of the excluded processes 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 (e.g. shared libraries) and we must ensure that excluding one process will not affect other processes when the VM is restored.

Davidoff et al. [12] retrieved clear text passwords from the physical memory of a Linux system. Their work aimed to show that the physical RAM may retain sensitive information even after the system has been powered off and the attacker with physical access to the system can steal information through cold boot memory dumping attacks. However, with checkpoints, the problem is significantly more severe: in the RAM, the amount of time the sensitive information persists in the memory after the machine is powered off is limited by the RAM's ability to retain information in absence of power. Also, the checkpoints are saved to the disk and the information stored in the checkpoints can persist for long time. In addition, they assume that the attacker has physical access to the system, but we do not.

6. CONCLUSIONS AND FUTURE WORK

This paper presents *SPARC*, a security and privacy aware VM checkpointing mechanism, which enables users to selectively exclude processes and terminal applications that contain users' confidential and private information from being checkpointed. *SPARC* helps minimize the lifetime of confidential information by preventing unintended checkpointing of process-specific memory contents. 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* poses only $1\% - 5.3\%$ of overhead with common application workloads.

In the future, we plan to extend *SPARC* to exclude confidential disk information from being checkpointed and develop techniques to exclude only particular data (e.g. the credit card information or specific emails) from being checkpointed, instead of excluding the entire process. The challenge is how to clear the data segment of a process without causing the process to crash after restoration. We will also investigate techniques to handle the case where excluding a process from being checkpointed may affect its parent/child processes and other processes that communicate with the process being excluded through e.g. files, sockets, and pipes. Such processes may receive information from the process being excluded and hence may contain sensitive information. We plan to detect such processes using the approach given in [49] and exclude them from being checkpointed if they are non-system-critical processes.

In addition, we will design VMs in which the state of each process is cleanly encapsulated. This would help avoid scrubbing process-specific information from disparate locations in OS memory. Another research avenue is to investigate process containers that can tightly isolate the entire state of a process and hence simplify the task of identifying and destroying sensitive information. Furthermore, we plan to implement *SPARC* as a kernel module, which will enable us to exclude processes without modifying the guest kernel.

Finally, *SPARC* assumes that the hypervisor and the VM have existing runtime protection mechanisms against malicious intrusions and focuses on exclusion of confidential process information from checkpoints. We will identify potential attacks that may specifically target *SPARC* to hide the attacker's activities and develop counter-measures.

7. REFERENCES

[1] Kernel based virtual machine. `http://www.linux-kvm.org/page/Main_Page`.

[2] Vmware ace virtualization suite. `http://www.vmware.com/products/ace/`.

[3] Xen hypervisor. `http://http://www.xen.org/`.

[4] D. P. Bovet and M. C. Ph. *Understanding the Linux Kernel, Third Edition*. O'Reilly Media, 3 edition, November 2005.

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

[6] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, pages 1–11, New York, NY, USA, 1995. ACM.

[7] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*, Seattle, WA, USA, Mar. 2008. ACM.

[8] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of USENIX Security Symposium*, pages 22–22, 2004.

[9] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum. Shredding your garbage: reducing data lifetime through secure deallocation. In *Proceedings of the USENIX Security Symposium*, pages 22–22, 2005.

[10] P. Chung, Y. Huang, S. Yajnik, G. Fowler, K.-P. Vo, and Y.-M. Wang. Checkpointing in cosmic: A user-level process migration environment. *Pacific Rim International Symposium on Fault-Tolerant Systems*, 0:187, 1997.

[11] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005.

[12] S. Davidoff. Cleartext passwords in linux memory. http://www.philosecurity.org, 2008.

[13] J. deok Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. In *In Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 48–59, 1998.

[14] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *15th ACM conference on Computer and communications security*, pages 51–62, 2008.

[15] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. 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.

[16] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '08, pages 121–130, New York, NY, USA, 2008. ACM.

[17] T. Garfinkel, B. Pfaff, J. Chow, and M. Rosenblum. Data lifetime is a systems problem. In *Proc. of ACM SIGOPS European workshop*. ACM, 2004.

[18] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: a virtual machine-based platform for trusted computing. pages 193–206. ACM Press, 2003.

[19] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium, pages*, pages 191 – 206, 2003.

[20] T. Garfinkel and M. 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.

[21] Google Corp. Inter-process communication. `http://dev.chromium.org/developers/design-documents/inter-process-communication`.

[22] A. Joshi, S. T. King, G. W. Dunlap, and P. 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.

[23] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. pages 1–15, 2005.

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

[25] O. Laadan and J. Nieh. Transparent checkpointrestart of multiple processes on commodity operating systems. In *In Proceedings of the 2007 USENIX Annual Technical Conference*, 2007.

[26] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '10, pages 155–166, New York, NY, USA, 2010. ACM.

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

[28] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department, April 1997.

[29] Microsoft Corp. Hyper-v server 2008 r2. `http://www.microsoft.com/hyper-v-server/en/us/overview.aspx`.

[30] A. M. Nguyen, N. Schear, H. Jung, A. Godiyal, S. T. King, and H. D. Nguyen. Mavmm: Lightweight and purpose built vmm for malware analysis. In *Annual Computer Security Applications Conference*, pages 441–450, 2009.

[31] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 124–131, 2009.

[32] D. A. S. d. Oliveira and S. F. Wu. Protecting kernel code and data with a virtualization-aware collaborative operating system. In *Annual Computer Security Applications Conference*, pages 451–460, 2009.

[33] Oracle Corp. Virtualbox. `www.VirtualBox.org`.

[34] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of zap: A system for migrating computing environments. In *In Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002*, pages 361–376, 2002.

[35] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *IEEE Symposium on Security and Privacy*, pages 233 – 247, 2008.

[36] P. R. G. r. A guide to understanding object reuse in trusted systems. `http://www.fas.org/irp/nsa/rainbow/tg018.htm`.

[37] R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *the 11th international symposium on Recent Advances in Intrusion Detection*, pages 1–20, 2008.

[38] N. Santos, K. P. Gummadi, and R. Rodrigues. Towards trusted cloud computing. In *HOTCLOUD*, 2009.

[39] A. Seshadri, M. Luk, N. Qu, and A. 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.

[40] D. A. Solomon and M. Russinovich. *Inside Microsoft Windows 2000*. Microsoft Press, 2000.

[41] J. Srouji, P. Schuster, M. Bach, and Y. Kuzmin. A transparent checkpoint facility on nt. In *in Proceedings of 2nd USENIX Windows NT Symposium*, pages 77–85, 1998.

[42] VMware. Cloud computing. `http://www.vmware.com/solutions/cloud-computing/`.

[43] VMware Inc. `http://www.vmware.com/`.

[44] VMware Inc. Vmware infrastructure. `http://www.vmware.com/landing_pages/discover.html`.

[45] R. J. Wysocki. Freezing of tasks. In *Linux Kernel 2.6.31 Documentation*, 2007.

[46] Xen. Xen cloud platform - advanced virtualization infrastructure for the clouds. `http://www.xen.org/products/cloudxen.html`.

[47] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th annual international symposium on Computer architecture*, ISCA '03, pages 122–135, New York, NY, USA, 2003. ACM.

[48] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam,

B. Weissman, and V. 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.

[49] H. Zheng and J. Nieh. Swap: a scheduler with automatic process dependency detection. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, pages 14–14, Berkeley, CA, USA, 2004. USENIX Association.