Privacy-preserving Virtual Machine

Tianlin Li *
Dept. of Computer Science
Binghamton University
Binghamton, NY, 13902
tli16@binghamton.edu

Yaohui Hu *
Dept. of Computer Science
Binghamton University
Binghamton, NY, 13902
yhu15@binghamton.edu

Ping Yang
Dept. of Computer Science
Binghamton University
Binghamton, NY, 13902
pyang@binghamton.edu

Kartik Gopalan
Dept. of Computer Science
Binghamton University
Binghamton, NY, 13902
kartik@binghamton.edu

ABSTRACT

Cloud computing systems routinely process users' confidential data, but the underlying virtualization software in use today is not constructed to minimize the exposure of such data. For instance, virtual machine (VM) checkpointing can drastically prolong the lifetime and vulnerability of confidential data without users' knowledge by storing such data as part of a persistent snapshot. A key requirement for minimizing the exposure of any data is the ability to cleanly isolate such data for either exclusion or processing. Traditional mechanisms for memory taint tracking are expensive whereas those for isolating application footprint in VM-based sandboxes are not transparent. In this paper, we propose a transparent and lightweight mechanism for isolating a confidential application's memory footprint in a VM. The key idea is for a parent VM to spawn a child VM, called a *Privacy-preserving Virtual Machine* (PPVM) within which the confidential application executes. Hypervisor features, such as VM checkpointing, that need to exclude the memory of a confidential application can safely ignore the child VM's memory footprint. Alternatively, features such as checkpoint encryption or malware tracking can operate only on the child VM's memory. We implement memory isolation for PPVM through a lightweight *VM fork* operation that uses copy-on-write to reduce the memory and filesystem overhead of the PPVM. Transparency is achieved through a *confidential shell* that allows the parent VM to spawn the confidential application in the PPVM and exercise control over it during runtime. We demonstrate the effectiveness of PPVM through its use with a standard hypervisor service, VM checkpointing, which can safely checkpoint the parent VM while excluding or encrypting the associated PPVM. We show that our PPVM implementation achieves effective memory isolation with low overheads on memory, CPU, and network performance.

*Co-first authors

ACSAC '15, December 07-11, 2015, Los Angeles, CA, USA
 ISBN 978-1-4503-3682-6/15/12$15.00
DOI: http://dx.doi.org/10.1145/2818000.2818044

1. INTRODUCTION

Modern cloud platforms increasingly process and store users' confidential data, such as passwords, financial information, health records, lawyer-client correspondence, and other personally identifying information. When using such cloud services, users have certain implicit expectations of data privacy, whether or not it is explicitly guaranteed by the cloud provider. Users may expect that their confidential information will not be stored beyond its useful lifetime; for example, credit card numbers will be forgotten after a successful transaction (unless explicitly authorized by the user), and passwords will not be stored in decrypted form except in memory during authentication. Users may also expect that their private data won't be disseminated in ways not reasonably expected by the user; for example, copies of their private data will not be duplicated and shared with third parties.

Cloud platforms heavily rely on virtualization technologies. The services hosted in the cloud often run within virtual machines (VMs). On one hand, VMs improve security in shared cloud infrastructures through greater isolation, and more transparent malware analysis and intrusion detection [31, 32, 37, 7, 9, 13, 19, 39, 35, 24]. On the other hand, VMs also give rise to new privacy and security challenges.

To handle confidential data correctly, any system must be able to first cleanly identify and isolate such data. We focus mainly on confidential data stored in the main memory since such data is more likely to be unencrypted. Unfortunately, the underlying virtualization and operating systems software in use today are not constructed to control and minimize the exposure of in-memory confidential data. Consequently, confidential data can be spread around in unexpected places in memory, increasing the risk of that data being stored and accessed beyond its useful lifetime and by unauthorized parties.

Existing OS-level approaches to minimizing the lifetime of confidential data include clearing deallocated memory [11, 6, 41, 1], taint tracking [38, 48], selective exclusion [15, 18], and selective encryption [11]; these mechanisms do not provide a single encapsulated memory space where the confidential data can be stored and processed, besides having high performance overheads. VM-based sandboxing approaches [10] tend to have high VM creation and memory overhead and may not be transparent to applications.

In this paper, we propose a transparent and lightweight mechanism for cleanly isolating a confidential application's memory footprint in a VM. The key idea is for a VM to spawn (or fork) a child

VM, called a *Privacy-preserving Virtual Machine* (PPVM), which runs alongside the original (parent) VM; confidential applications execute in the PPVM whereas regular applications run in the parent VM. Hypervisor features, such as VM checkpointing, that need to exclude the memory of a confidential application can safely ignore the PPVM's memory footprint. Alternatively, features such as checkpoint encryption or malware tracking can selectively operate only on the PPVM's memory.

PPVM has two salient features that make it different from a traditional VM. First, PPVM is *lightweight*, which means that PPVM can be launched quickly and does not impose significant overheads beyond the requirements of the confidential application that it runs. We implement a lightweight *VM fork* operation that uses copy-on-write to reduce the memory and filesystem overhead of the PPVM. Second, PPVM is *transparent* to the confidential applications, which means that confidential applications are unaware that they are running within a PPVM. Transparency is achieved through a *confidential shell* that allows the parent VM to spawn the confidential applications in the PPVM and control them during runtime.

We demonstrate the effectiveness of PPVM using a standard hypervisor service, VM checkpointing, which saves a persistent snapshot of a VM's state at a given instant. A VM's state includes, at the minimum, its memory image and CPU execution state and possibly additional state such as virtual disk contents. The checkpoint 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 [44], Hyper-V [28], VirtualBox [34], KVM [22], and Xen [47] support VM checkpointing. Despite the above benefits, VM checkpoints can drastically prolong the lifetime and vulnerability of sensitive data. Data that should normally be discarded quickly after processing, such as passwords, credit card numbers, health records, or lawyer-client conversations can now be saved forever in persistent storage through VM checkpointing [15]. We show that when a VM executes its confidential applications in a PPVM, a VM checkpointer can easily save the VM's memory snapshot while selectively excluding (or encrypting) the contents of any confidential applications running in the PPVM. Our evaluations show that PPVM achieves effective memory isolation for confidential applications with low overheads on memory, CPU, and network performance.

Assumptions and threat model: We assume that the VM and its user applications are not compromised. We also assume that the hypervisor and its checkpointing mechanism are secure. We also assume that anytime after a VM is checkpointed, multiple third parties (including malicious attackers) have the ability to access the checkpoint, examine it, or copy it to a remote site for later analysis. The third parties can also restore the VM from the checkpoint or modify the checkpoint. Finally, the checkpoint may be distributed or shared among multiple unknown users.

The rest of the paper is organized as follows. Sections 2 and 3 present the design and implementation of PPVM. The experimental results are presented in Section 4. Section 5 describes the related work and Section 6 concludes the paper.

2. DESIGN OF PPVM

We propose a *privacy-preserving VM* (PPVM) to facilitate clean and exhaustive identification of a confidential application's memory footprint and disk contents, which can then be excluded (or optionally encrypted) by the checkpointer. Figure 1 illustrates the architecture of the PPVM. A PPVM is a special lightweight VM which executes alongside a regular VM (called "Primary VM") that is to be checkpointed. Confidential applications, which would normally execute within the Primary VM as "regular" processes, are now executed within the PPVM as one or more "PPVM processes". When the Primary VM is checkpointed, the entire memory footprint of the PPVM is excluded from the checkpoint. PPVM includes all the system services needed by confidential processes. Thus the memory footprint and disk contents of confidential processes are cleanly isolated from the Primary VM. We no longer need to track the confidential application's system-wide dependencies, such as its scattered memory footprint or I/O operations. When the Primary VM is restored from a checkpoint, the restored image would be automatically free of any confidential processes that existed before checkpointing.

The two requirements in realizing PPVMs are that (1) PPVM must be *lightweight*, and (2) PPVM must be *transparent* to the confidential applications being excluded from the checkpoint. In the rest of this section, we describe how the above two requirements are addressed in the PPVM design.

2.1 Lightweight

The requirement of being lightweight means that PPVM must be launched quickly and should not impose significant memory, CPU, and I/O overheads beyond the requirements of the confidential applications that it runs. Launching a PPVM involves copying the CPU and I/O states from the Primary VM to the PPVM and enabling PPVM to access identical memory and disk content as the Primary VM has at launch time, but without any copying overheads. PPVM uses copy-on-write (COW) sharing of memory with the Primary VM, much like process fork and VM Fork operations [46, 16, 26]. Similarly, if PPVM's disk I/O operations are considered confidential, then PPVM can optionally share a copy-on-write disk image with the Primary VM. Thus, upon launch, PPVM is essentially a clone of the Primary VM insofar as its critical system components are concerned. Copies are subsequently made only for those memory and disk contents that are dirtied by either the PPVM or the Primary VM.

2.2 Transparency

The transparency requirement means that confidential applications should be unaware that they are running within a PPVM. In other words, all PPVM processes appear to be part of the Primary VM's system image, and observe exactly the same process and I/O namespace as other processes in the Primary VM. However the PPVM provides a self-contained execution context for confidential processes, including separate OS kernel, libraries, networking, I/O, and other services. Transparency of the PPVM is achieved through a *confidential shell* that allows a user to use the Primary VM to spawn the confidential application in the PPVM and to observe and control the execution of PPVM processes the same way she can control regular processes in the Primary VM. We use Agents, which are user-level programs, to communicate between the Primary VM and the PPVM. For instance, the Primary VM's agent conveys the the path and the arguments for executing the confidential application to the PPVM's agent. At the time of restoring the Primary VM from the checkpoint, the hypervisor uses an event-channel based callback mechanism to inform the guest OS in the Primary VM of the restoration operation. The callback resets the guest OS' internal state to account for the absence of the PPVM.

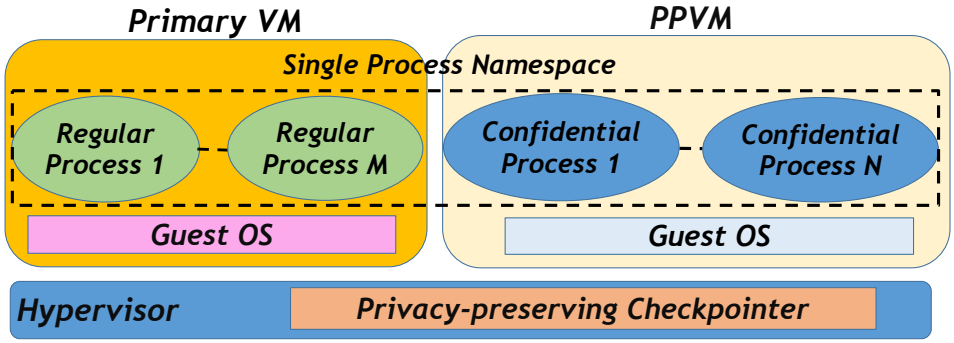


Figure 1: PPVM Architecture: Primary VM and PPVM share copy-on-write memory. PPVM's confidential processes remain under the control of the Primary VM's root.

3. PPVM IMPLEMENTATION

We have implemented PPVM in QEMU/KVM virtualization platform in the Linux operating system. This section presents the implementation details of our system.

3.1 Memory Copy-on-write

KVM uses extended page table (EPT) to implement memory virtualization. When a process inside a VM accesses a memory location, the guest page table is used to translate the guest virtual address of the memory into the guest physical address. EPT is then used to translate the guest physical address to the host physical address. If the guest physical address is not in the EPT, then the VM exits and KVM uses the host page table to get the correct page frame number and adds the mapping to the EPT. The process inside the VM can access the memory on the host once the corresponding host physical page is obtained from EPT.

The key insight behind making PPVM lightweight is implementing a memory copy-on-write mechanism, which means that memory pages of both VMs are mapped to the same page frames and these page frames are marked copy-on-write. To achieve this, we first mark the host page table and the EPT entry of the Primary VM as read-only. We then build the host page table for PPVM, which maps the host virtual address of the PPVM memory to the corresponding host physical address (page frame number) used by the Primary VM; the page map count of the corresponding page frame is increased by 1. We wrote a KVM ioctl API VM_Fork to implement the quick cloning of the Parent VM. We also implemented two functions Set_RAM and $Record\_Child$ which were used to record the information needed for implementing VM_Fork. Set_RAM is invoked when the multiple memory regions, called RAM blocks, are initialized in QEMU. Set_RAM stores the information about the RAM blocks, such as slot number, RAM size, and the starting host virtual address, in the kernel. $Record\_Child$ records the data structures used by the PPVM, including pointers to $struct\ kvm$ (a data structure storing the information of a VM such as vCPU and memory etc.) and $struct\ mm\_struct$ (a data structure storing the memory information of processes).

Our first implementation of VM_Fork was to construct the host page table entries for PPVM one at a time, as shown in Figure 2. VMs with the same configuration have the same physical memory layout and the same KVM memory slot layout. KVM memory slot structure records the physical memory layout in a VM, which stores the guest frame number and the corresponding host virtual address. Figure 3 shows how to compute the host virtual address

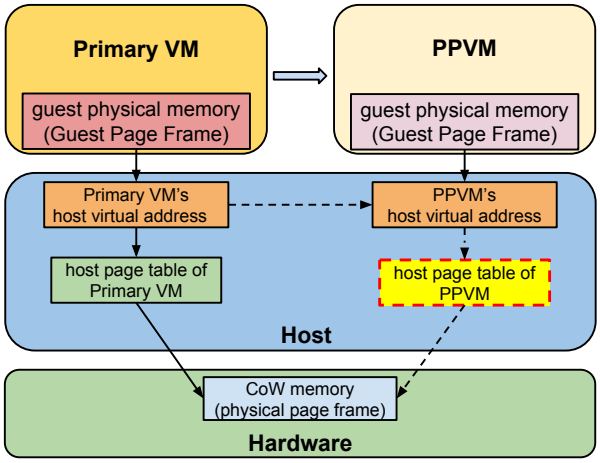


Figure 2: An implementation of VM_Fork that constructs each host page table entry for PPVM individually.

of the PPVM from the host virtual address of the Primary VM. For every host virtual address in the Primary VM, we get the KVM slot ID and offset of the corresponding RAM block; this information is obtained when the RAM block is added to both VMs. We then use the same slot ID and offset to get the host virtual address of the PPVM from PPVM's RAM block. Next, we obtain the page frame number from the Primary VM's host page table. With the PPVM's host virtual address and page frame number, we build the corresponding host page table entry for the PPVM.

The above approach, however, is inefficient: To construct each page table entry of the PPVM, we need to traverse multi-level page tables – page global directory(pgd), page upper directory(pud), page middle directory(pmd), and the host page table of the Primary VM. Our experimental results show that, it took around 25 seconds to process memory copy-on-write for an idle VM with 1GB memory size. Inspired by the normal fork mechanism, we leverage the existing kernel function copy_page_range() to create the PPVM page table in batches (as shown in Figure 4), which significantly reduces the number of times it takes to traverse the multi-level page tables. In the normal process fork mechanism, after setting up the process descriptor, copy_mm() function is called to create new page tables and copy the contents of page table entries

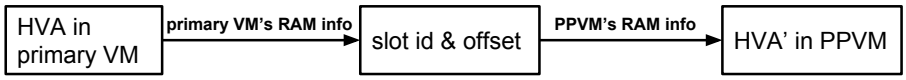


Figure 3: Computing the host virtual address in PPVM from the host virtual address of the Primary VM.

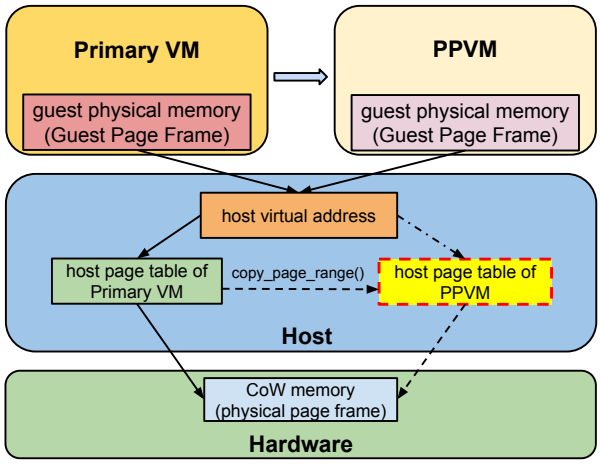


Figure 4: A more efficient implementation of VM_Fork that constructs host page table entries for PPVM in batches.

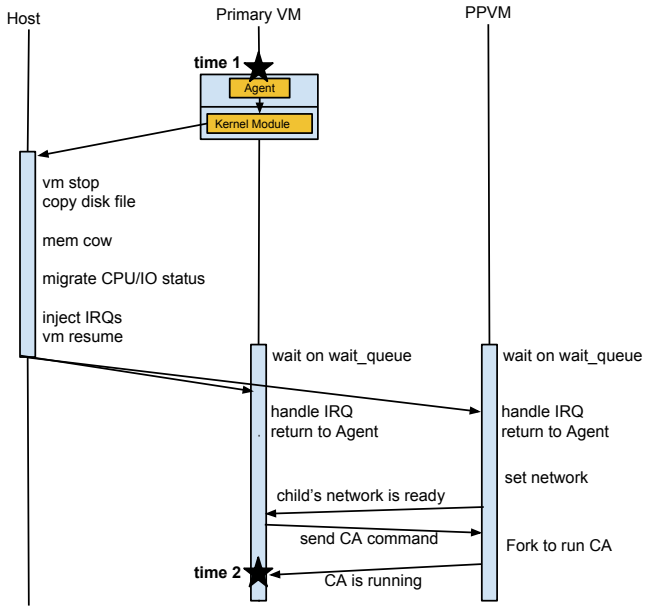


Figure 5: Sequence of events in forking a PPVM from the parent VM and launching a confidential application.

from the parent process to the child process. Since both the parent and child processes have identical process address space, the child process does not need to traverse the multi-level page tables every time when constructing a page table entry. In our VM_Fork implementation, we modified the function to initialize RAM blocks to force the PPVM to have the same host virtual address as the Primary VM for every region of virtual memory. We then copy all the page entries from the Primary VM in batches within the virtual address range. The above optimization reduces the memory COW time from 25 seconds to less than 100 milliseconds.

3.2 Disk Copy-on-Write

To prevent the confidential data from being stored on the disk when the Primary VM is checkpointed, the PPVM has the option to be configured with its own disk space. Upon PPVM's launch, the PPVM and its Primary VM share the same copy-on-write disk image. After the confidential application runs inside the PPVM, any I/O operations performed by the confidential application are directed to PPVM's own disk.

In QEMU, we use disk snapshot to start the VM. The snapshot in QEMU refers to the base image and Redirect-on-Write is applied everytime when the VM is checkpointed to avoid modification to the base image. To enable the PPVM to have the same disk contents as the Primary VM before the PPVM starts, we can make a copy of the disk snapshot of the primary VM and use the copy as the disk file image for the PPVM. However, copying the whole snapshot could be time-consuming if the size of the snapshot is large. For example, our experimental results show that it took around 35 seconds to make a copy of a snapshot whose size is 2.9GB. Alternatively, we can also commit the snapshot of the primary VM to the base image just before PPVM starts (using qemu-img commit) and then create a second snapshot of the base image as the disk file for the PPVM (using qemu-img create). However, committing the snapshot may be time consuming. For example, in our experiment, it took around 25 seconds to commit a snapshot whose size is 2.9GB and 0.1 second to create a new snapshot.

To reduce the launch time of PPVM, we use B-tree file system (Btrfs) to improve the efficiency of creating the initial disk image for the PPVM. Btrfs is a copy-on-write file system. It provides a cloning feature which creates a new inode sharing the same disk blocks with the original files. The data blocks are not duplicated. Since Btrfs uses copy-on-write, any modifications to the cloned file are not visible to the original file and vice versa. By putting the base image and the snapshots under the directory in which Btrfs is installed, we just need to clone the snapshot of the Primary VM to create the initial disk image for the PPVM. Our experimental results show that, with Btrfs, it took less than 150 milliseconds to clone a snapshot whose size is 2.9GB.

3.3 Launching the PPVM

In PPVM, we use agents to start the PPVM automatically and to communicate between the Primary VM and the PPVM. Figure 5 shows the sequence of steps for launching the PPVM. First, the user requests the agent in the Primary VM to start a confidential

application. Since the agent is a user-level program in the VM and VM_Fork is implemented in the QEMU process on the host, the agent cannot directly call the VM_Fork function. To allow the agent to trigger VM_Fork, we developed a guest kernel module with character device interface that provides ioctl API for agent to trap into the kernel. The module then uses UDP connection to inform QEMU to start PPVM and waits on a wait queue. The QEMU process of the Primary VM has a separate thread waiting for the VM_Fork request. This thread stops the Primary VM, creates the disk image for the PPVM, performs memory copy-on-write, and migrates CPU and I/O states of the Primary VM to the PPVM. After VM_Fork is complete, there are two QEMU processes, one for the Primary VM and another for the PPVM. Each QEMU injects a different interrupt to wake up their corresponding VMs which are waiting in the queue. Based on the interrupt handler invoked, each VM can determine whether it is the Primary VM or the PPVM. The agent then returns from the kernel module to the user space with the return value specifying whether it is the Primary VM or the PPVM. Based on the different return values, the agents can act on behalf of their corresponding VMs, much like how the return value from a regular `fork()` system call distinguishes between a parent and a child process. The agent in the PPVM then sets up the network and notifies the Primary VM that it is ready to run the confidential application. Finally, the primary VM sends the application path and parameters needed to execute the application to the PPVM. The PPVM then forks a new process to run the application and notifies the Primary VM.

3.4 Network Transparency

While PPVM provides all necessary system services in a self-contained manner, some services would need to mirror or share the Primary VM's services. For example, network transparency requires that confidential applications which run within PPVM, must be able to use the network identity of the parent VM when communicating with external hosts. Specifically, the PPVM must use the same IP and MAC addresses as the Primary VM. To achieve this, we developed an IP/MAC mirroring functionality in the underlying hypervisor's network subsystem. The PPVM is configured with a new MAC address and a new IP address, which is in the same subnet as the Primary VM. We then use a NAT layer on the host to forward the inbound packets to the proper VM based on the port number and to change the IP address of outbound packets to a common IP address for both VMs. For example, assume that the common IP address is IP_c which is an IP alias of the host, the IP address of the Primary VM is IP_1, and the IP address of PPVM is IP_2. When a user sends a TCP request (IP_c,Pn) to the host machine where the PPVM resides, requesting to connect to a server running on the PPVM that has port number Pn, the host intercepts the request and change the request to (IP_2,Pn) based on the NAT rules set previously. When the Primary VM or the PPVM sends requests to an external machine, the IP address is changed to IP_c.

3.5 Display service

Confidential applications in PPVM may need display services, such as a `tty` terminal or a graphical window. This is challenging because we wish to avoid tainting the parent VM's memory with child VM's display contents. Currently, we use SSH with X11 forwarding to provide remote login and display access to the PPVM. After the PPVM sets up the network, The agent issues command "ssh -XC root@⟨ internal IP of PPVM ⟩ ⟨ application ⟩", which enables the remote execution of commands inside the PPVM. In the future, we will investigate hypervisor-supported mechanisms that allow a PPVM to access the system display without requiring any displayed data to pass through the Primary VM.

4. EXPERIMENTAL RESULTS

In this section, we evaluate the effectiveness, launch time, and runtime overheads of our PPVM prototype. All experiments were conducted on a host system with 24 Intel(R) Xeon(R) CPU 2.10GHz processor and 128GB of RAM, and running Debian GNU/Linux 7.6 kernel version 3.17.2.

4.1 Effectiveness of memory isolation

We first demonstrate that PPVM successfully isolates the memory footprint of confidential applications for clean exclusion by the checkpointing service in the hypervisor. We used the Firefox and Chrome browsers, the MySQL database, and Gedit text editor as sample confidential applications.

MySQL database server: We ran the MySQL server in a regular VM and connected to the MySQL server from an external machine. Executing the query "select * from employee;" gave the following output:

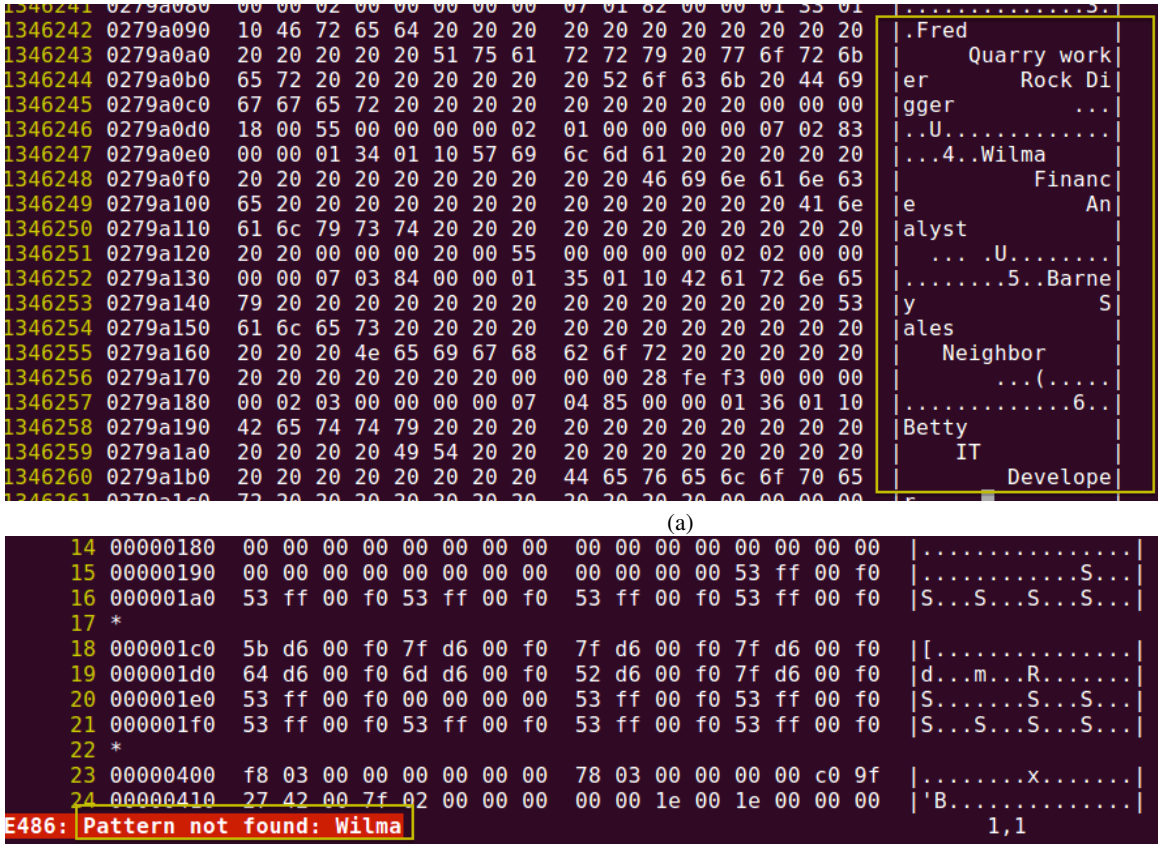
Name	Dept	Jobtitle
Fred	Quarry worker	Rock Digger
Wilma	Finance	Analyst
Barney	Sales	Neighbor
Betty	IT	Developer

Next, we checkpointed the regular VM. As shown in Figure 6(a), the snapshot contains the above contents listing all the users. We then ran the MySQL server as a confidential application in a PPVM by launching it via a confidential shell. We executed the same query and then checkpointed the VM. As shown in Figure 6(b), the confidential data about the employees could not be found in the snapshot.

Firefox, Chrome, and Geditor: We first ran Firefox in the Primary VM, entered URL "http://google.com", typed string "strp-pvmabc" in Google search engine, and checkpointed VM. Upon searching the binary snapshot, the string appeared in the snapshot 1086 times. We then used the confidential shell to launch Firefox in the PPVM, entered the same string in Google search engine, and checkpointed the Primary VM. As expected, the string was not found in the snapshot. Similar exclusion behavior was verified with Chrome and Geditor.

4.2 Memory Overhead of PPVM

We wrote a simple memory-write-intensive application, which repeatedly writes random numbers to 64MB memory sequentially, to evaluate the memory overhead of PPVM. Figure 7 gives the total amount of memory used by a single VM, PPVM implemented with copy-on-write (COW), and PPVM implemented with VM replication (using pre-copy VM migration algorithm), when the memory-write-intensive application runs inside either the VM or the PPVM. In this experiment, all VMs are configured with 2 vCPUs and 4GB memory. We measured Unique Set Size(USS) and Proportional Set Size(PSS) using $smem$ tool [40]. USS is the amount of unshared memory that uniquely belongs to a process. PSS is the amount of unshared and proportional shared memory (a proportion of the shared memory divided by the total number of processes that share the memory). The unique memory of the Primary VM and the



(a)

(b)

Figure 6: Memory snapshots when running the MySQL server in (a) Regular VM and (b) PPVM.

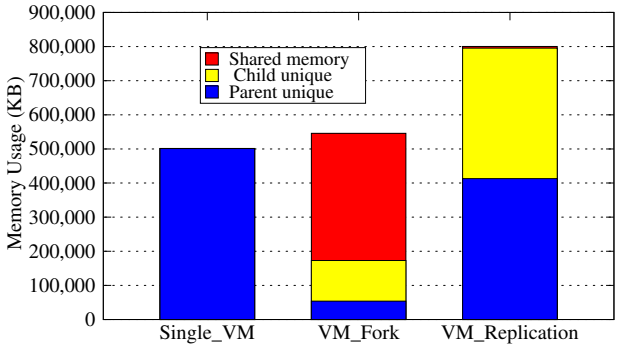


Figure 7: Memory usage of a single VM, PPVM with COW, and PPVM with VM replication.

PPVM is measured via USS directly. The memory shared by the Primary VM and the PPVM is measured as " (Primary VM's PSS - Primary VM's USS) + (PPVM's PSS - PPVM's USS)". Our experimental results show that, with copy-on-write, the memory usage of the Primary VM and the PPVM together is slightly (less than 10%) higher than that of a single VM. VM replication using precopy migration imposes around 60% overhead.

4.3 Launch Time of PPVM

Tables 1 and 2 give the time spent at each step to launch a PPVM with different memory usage and vCPUs, respectively. The application running inside the PPVM is a simple program that prints a sequence of characters. The launch time of the PPVM is computed as $time2 - time1$ in Figure 5. For both tables, the VMs are configured with 4GB memory and the disk snapshot image size is around 240MB. Before launching the PPVM, the Primary VM does not run any other applications except the agent. In Table 1, the memory used by the Primary VM (measured with "free -m") ranges from 250MB to 2GB, and the VM is configured with two vCPUS. In Table 2, the memory used by the Primary VM is fixed (around 250MB), and the Primary VM is configured with different number of vCPUs.

To control the memory usage, we create a *tmpfs* file system mounted on directory /mnt/tmp. We then use *dd* command to create files under /mnt/tmp with specific size. All files created under that directory will be stored in the main memory. Before launching the confidential application, we use "free -m" to check the memory usage.

Our experimental results show that the VM stop time, CPU and I/O migration time, and network set up time in the PPVM are relatively stable regardless of the size of memory usage in the Primary VM and the number of vCPUs. However, the memory copy-on-write time increases linearly when the memory usage of the Primary VM increases because it takes longer time to reconstruct the host page table entries for the PPVM (Figure 4). We also observed that the VM resume time increases linearly when the number of vCPUs increases. This is because vCPUs are implemented using

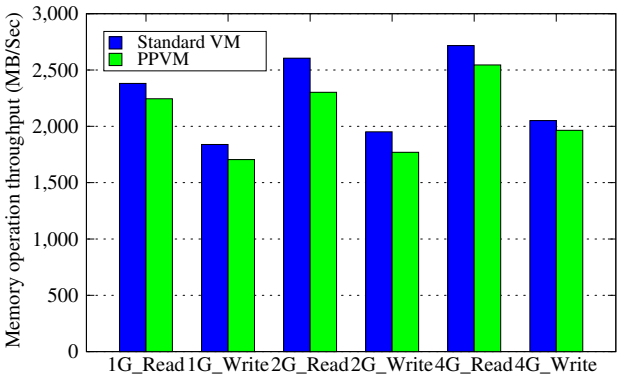


Figure 8: Comparison of memory read/write throughput between a single VM and the PPVM.

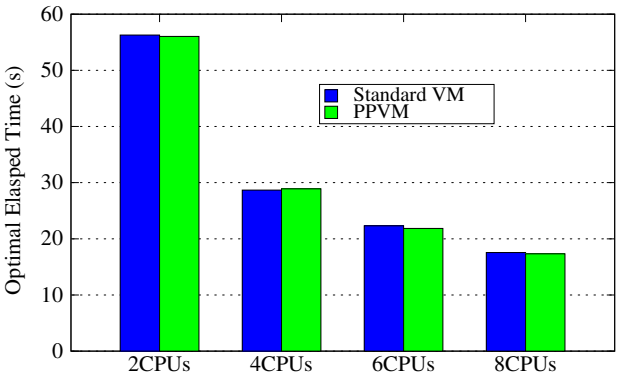


Figure 9: Comparison of kernel compilation time using Kernbench between a single VM and the PPVM.

threads in QEMU and the operation to resume all vCPUs individually activates each thread.

4.4 Application Performance Overhead

This section compares the memory, CPU, and network performance of applications running inside the standard VM and the PPVM.

Memory read/write performance.

We used *sysbench*[42], a modular benchmark tool, to get the performance of memory read/write operations. *sysbench* first allocates a memory block of a specified size and then performs read/write operations on the allocated memory. The above process is repeated until the total buffer size that has been read from or written to reaches total memory size specified by the user. In our experiments, VMs are configured with 2GB memory and 2 vCPUs. Memory block size is 1KB. The read/write operations are performed sequentially. The number of threads is 1. Figure 8 compares the memory read/write throughput with different total memory sizes (1GB, 2GB, and 4GB). The results show that the memory read/write throughput is slightly less in PPVM than in the standard VM. This is because after VM fork, the extended page table (EPT) is empty in PPVM initially and memory copy-on-write leads to more page allocations for write operations.

CPU performance.

We used Kernbench [23] to compile Linux kernel 3.18.1 inside the standard VM and the PPVM and compare the compilation time. Both the standard VM and PPVM are configured with 4GB memory and various vCPUs. As shown in Figure 9, with different numbers of vCPUs, it takes almost the same time to compile the kernel in the standard VM and in the PPVM.

Network performance.

We used Netperf [29] to measure the network bandwidth in PPVM by running the Netperf client inside the PPVM and running the Netperf server on an external machine. The Netperf client starts after the PPVM starts. We measured network bandwidth for the host machine, the standard VM, the Primary VM, and the PPVM, using options TCP_STREAM (netperf server receiving data from the client) and TCP_MAERTS (netperf server sending data to the client). We also measured the TCP request/response performance using TCP_RR option, performance of opening/closing TCP connection using TCP_CC option, and the TCP connect/request/response performance using TCP_CRR option. The request/response performance is measured as "transactions/sec", which specifies the number of exchanges of requests and responses every second. Both standard VM and Primary VM are configured with 2GB memory and 2 vCPUs. The standard VM uses unmodified QEMU and KVM code, while the Primary VM uses our modified QEMU and KVM code. As Table 3 shows that, compared to a standard VM, the Primary VM and the PPVM impose negligible overhead on the network bandwidth, about 20% overhead on TCP_RR and TCP_CC tests, and about 25% overhead on TCP_CRR test. The latter overheads are related to the use of NAT for network transparency in PPVM.

5. RELATED WORK

Previous work on *minimizing data lifetime* and *object reuse* has focused on clearing the deallocated memory (also known as *memory scrubbing*) [11, 6, 41, 1]. However, memory scrubbing does not solve the problem of confidential data being checkpointed *before* the pages are deallocated. Garfinkel et al. [12] 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 the checkpoint has also been recommended in [14, 45]. However, selective encryption of confidential application's memory footprint in the checkpoint is harder because of difficulties in exhaustively tracking an application's footprint in most modern operating systems. Additionally, encryption alone is not enough if the data should have been quickly erased by the application after use (such as passwords, or credit card numbers), but wasn't, or if such data was lying around in deallocated memory pages that should have been scrubbed by the OS, but was not for performance reasons. If the VM is restored from an encrypted checkpoint at some arbitrary point in the future, the confidential data will be decrypted and loaded into the memory, thus exposing such data again.

Work in [11] has proposed explicitly encrypting confidential data in the memory and clearing such data after use by discarding the key. However, since the confidential data remains part of the VM's memory, such data may still be exposed if VM checkpointing occurs just after a program decrypts the data for use. Anonymous

Table 1: The launch time of PPVM with different memory usage. Primary VM is configured with two vCPUs and 4GB maximum memory.

Memory usage (MB)	VM stop (ms)	Disk copy (ms)	COW (ms)	Migrating CPU/IO (ms)	VM resume (ms)	Network setup (ms)	Launch time (ms)
250	11.17	106	66.67	0.83	71.67	88.67	484.05
500	16.17	107	99.17	1	68.5	89.33	519.78
750	12.43	92.57	133.57	1	73.71	93.43	562.83
1000	15.67	94.33	181.5	1	70.83	94.17	615.29
1500	12.17	106.5	230.34	1	81.17	99.17	680.75
2000	15.29	101.29	284.57	1	66.71	96.57	727.67

Table 2: The launch time of PPVM with different number of vCPUs. Primary VM is configured with 4GB maximum memory and 250MB actual memory usage.

vCPU	VM stop (ms)	Disk copy (ms)	COW (ms)	Migrating CPU/IO (ms)	VM resume (ms)	Network setup (ms)	Launch time (ms)
2	11.17	106	66.67	0.83	71.67	88.67	484.05
4	12	107.83	61.5	1.33	164	102.33	694.90
6	12	107	74.67	1.5	232.67	111	855.15
8	16.17	111.83	62.5	1.67	329.83	107	1051.18

execution mode for applications has been proposed in [10] and [33] so as to hide traces of an application's execution after it terminates. However, such approaches do not hide the confidential data in memory during the execution of applications and hence do not prevent such data from being checkpointed. Chen et al. [4] proposed a VM-based system called Overshadow to protect confidential application data. Overshadow enables secure execution of applications even if the guest OS is compromised. To do so, Overshadow presents different views of application's memory so that an application can access memory storing the application's data, but the guest OS can only access the encrypted data. However, this approach, does not prevent the confidential application data from being checkedpointed by the hypervisor if VM checkpointing occurs just after the application decrypts the data for use.

All the above approaches share a common weakness that the confidential application's memory remains part of the VM being checkpointed. In contrast, PPVM is designed to explicitly isolate the confidential application's memory footprint, so it becomes easier to exclude such data during checkpointing.

Gofman et al. [15] developed techniques to exclude confidential data from being checkpointed. However, their approach had to examine disparate memory locations in the VM's kernel, the virtual memory of the application, deallocated pages, socket/pipe/FIFO/TTY buffers, device driver memory etc.. In contrast, PPVM helps avoid scrubbing process-specific information from disparate locations. Hu et al. [18] 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, requires the programmer to specify the location of confidential data using application programmer interfaces (APIs), and hence is not application-transparent. Ta-Min et al. [43] proposed to partition system calls into trusted and untrusted. Untrusted system calls are handled by the commodity OS, while the trusted system calls are handled by the private OS. Doing so requires the modification to applications. ARM TrustZone technology [2] proposed "secure mode", a special CPU mode for providing a trusted execution environment with Cortex-A processors support. By activating the secure mode, applications can access physical resources that are hidden in the non-secure mode. This approach also requires the modification of applications.

A number of researchers have developed memory taint analysis techniques to identify all memory locations storing the confidential data. However, traditional memory taint analysis systems [5, 17, 30, 48] have a high performance overhead for large applications due to the use of binary emulation or interpretation. Additionally, even with taint tracking, selectively and safely excluding or encrypting the confidential application data from a checkpoint is extremely difficult requiring, from our experience, extensive changes to the core operating system.

A recent trend is the use of Process Containers [8, 27, 20] as a lightweight alternative to full system VMs. Containers provide isolated execution for one or more user-level processes. The main objective of containers is to restrict the resource usage, namespace, and certain capabilities of the contained processes in order to protect the host system and other applications, such as through a separate filesystem, process address space, and superuser. However, containers do not isolate the memory footprint of the contained processes from the host system, the way PPVM does. Unlike containers, a PPVM does not restrict the capabilities of its contained processes, except to control data transmission back to the parent VM. Unlike processes in containers, PPVM processes share the process namespace and all other capabilities available to processes in the Primary VM. For instance, processes in a PPVM are under the administrative control of the superuser of the Primary VM and share the filesystem and network identity with other processes in the Primary VM.

Prior research has also considered checkpointing and replaying process execution as a means for intrusion detection, debugging, process migration, and fault tolerance [3, 9, 21, 25]. However, none of them examine the data lifetime implications of checkpointing.

Techniques for *VM Fork* that use copy-on-write for rapid VM cloning have been explored in prior research including projects such as SnowFlock [26] and Fargo [36]. We implemented our own

Table 3: Network bandwidth and request/response/connect performance.

	Network Bandwidth (Mbps)		Request/Response/Connect (Transaction/s)		
	TCP_STREAM	TCP_MAERTS	TCP_RR	TCP_CC	TCP_CRR
Host machine	941.33	941.36	8120.38	3880.81	3375.56
Standard VM	940.82	941.13	4841.56	2474.50	2004.31
Primary VM	939.63	940.62	4059.82	1983.79	1625.74
PPVM	939.35	940.63	4062.66	1963.62	1618.18

VM Fork mechanism in KVM/QEMU because no corresponding implementation of VM Fork existed in the KVM/QEMU platform that used copy-on-write to perform rapid VM cloning. Additionally, to the best of our knowledge, no prior work has investigated the specific optimizations that we introduce to speed up the memory and disk copy-on-write mechanism during a VM Fork, nor have they examined the sharing of network identity between the parent and child VMs and transparently controlling child VM's processes from the parent.

6. CONCLUSION

Virtualization technologies used in cloud platforms provide inadequate support to identify and isolate the memory footprint of specific applications. Consequently, services that need to process or exclude confidential data, such as VM checkpointing or encryption, are less effective. In this paper, we proposed a *privacy preserving virtual machine* (PPVM) which facilitates clean and exhaustive identification of a confidential application's memory footprint. PPVM is spawned by a parent VM using a lightweight *VM Fork* operation which uses copy-on-write to reduce memory and filesystem overheads on the host system. Confidential applications are executed within the PPVM, but transparently controlled by the parent VM via a confidential shell. We demonstrate the effectiveness of PPVM via a privacy-preserving checkpointing mechanism which can safely exclude or encrypt the PPVM's memory while saving a snapshot of the parent VM's memory. Evaluations show that our PPVM implementation achieves effective memory isolation with low overheads on memory, CPU, and network performance.

Acknowledgment This work is supported in part by the National Science Foundation through grants CNS-0845832, CNS-1320689, and CNS-1527338.

7. REFERENCES

[1] J. P. Anderson and R. Vaughn. A guide to understanding object reuse in trusted systems. Technical report, DTIC Document, 1992.

[2] ARM Limited. ARM Security technology: Building a Secure System using TrustZone Technology. ARM White Paper PRD29-GENC-009492C.

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

[4] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proc. of ASPLOS*, 2008.

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

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

[7] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware analysis via hardware virtualization extensions. In *Proc. of CCS*, pages 51–62, 2008.

[8] Docker Inc. *https://www.docker.com/*.

[9] 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 *Proc. of OSDI*, 2002.

[10] A. M. Dunn, M. Z. Lee, S. Jana, S. Kim, M. Silberstein, Y. Xu, V. Shmatikov, and E. Witchel. Eternal sunshine of the spotless machine: Protecting privacy with ephemeral channels. In *Proc. of OSDI*, 2012.

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

[12] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proc. of SOSP*, 2003.

[13] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. of NDSS*, 2003.

[14] T. Garfinkel and M. Rosenblum. When virtual is harder than real: Security challenges in virtual machine based computing environments. In *Proc. of HotOS*, 2005.

[15] M. I. Gofman, R. Luo, P. Yang, and K. Gopalan. SPARC: A security and privacy aware virtual machine checkpointing mechanism. In *Proc. of the ACM Workshop on Privacy in the Electronic Society (WPES)*, 2011.

[16] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. *Communications of the ACM*, 53(10):85–93, 2010.

[17] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection using demand emulation. In *EuroSys*, 2006.

[18] Y. Hu, T. Li, P. Yang, and K. Gopalan. An application-level approach for privacy-preserving virtual machine checkpointing. In *The 6th IEEE International Conference on Cloud Computing, research track*, 2013.

[19] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proc. of the ACM SOSP*, 2005.

[20] P. Kamp and R. N. M. Watson. Jails: Confining the omnipotent root. In *Proc. 2nd Intl. SANE Conference*, 2000.

[21] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proc. of USENIX Annual Technical Conference*, pages 1–15, 2005.

[22] A. Kivity, Y. Kamay, and D. Laor. kvm: the Linux Virtual Machine Monitor. In *Proc. of Ottawa Linux Symposium*, 2007.

[23] C. Kolivas. *Kernbench: http://ck.kolivas.org/apps/kernbench/kernbench-0.50/.*

[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, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proc. of ACM SIGMETRICS*, 2010.

[26] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan. SnowFlock: rapid virtual machine cloning for cloud computing. In *Proc. of Eurosys*, 2009.

[27] Linux Containers. *https://linuxcontainers.org/.*

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

[29] Netperf. http://www.netperf.org/netperf/.

[30] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Network and Distributed System Security Symposium (NDSS)*, 2005.

[31] 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.

[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] K. Onarlioglu, C. Mulliner, W. Robertson, and E. Kirda. Privexec: Private execution as an operating system service. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, 2013.

[34] Oracle Corp. Virtualbox. www.VirtualBox.org.

[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] Project Fargo. http://www.yellow-bricks.com/2014/10/07/project-fargo-aka-vmfork-what-is-it/.

[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] P. Saxena, R. Sekar, and V. Puranik. Efficient fine-grained binary instrumentationwith applications to taint-tracking. In *Proceedings of International Symposium on Code Generation and Optimization*, 2008.

[39] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *ACM SIGOPS Operating Systems Review*, volume 41(6), pages 335–350. ACM, 2007.

[40] Smem memory reporting tool. http://selenic.com/smem.

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

[42] Sysbench. https://wiki.gentoo.org/wiki/sysbench.

[43] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proc. of Operating Systems Design and Implementation*, 2006.

[44] VMware Inc. http://www.vmware.com/.

[45] VMware Inc. VMWare Ace Virtualization Suite. http://www.vmware.com/products/ace/.

[46] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. Snoeren, G. Voelker, and S. Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. *SIGOPS Operating Systems Review*, 39(5):148–162, 2005.

[47] Xen Hypervisor. http://http://www.xen.org/.

[48] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. Tainteraser: Protecting sensitive data leaks using application-level taint tracking. *SIGOPS Oper. Syst. Rev.*, 45(1):142–154, Feb. 2011.