CRIMES: Using Evidence to Secure the Cloud

Sundaresan Rajasekaran
George Washington University
sundarcs@gwu.edu

Harpreet Singh Chawla
George Washington University
harpreetsc@gwu.edu

Zhen Ni
George Washington University
leonizhen@gwu.edu

Neel Shah
George Washington University
nshah95@gwu.edu

Emery Berger
University of Massachusetts Amherst
emery@cs.umass.edu

Timothy Wood
George Washington University
timwood@gwu.edu

ABSTRACT

Cloud applications are appealing targets to attackers, yet current cloud infrastructures have few ways of helping defend their customers from attacks. However, the use of virtual machines, and the economy of scale found in cloud platforms, provides an opportunity to offer strong security guarantees to tenants at low cost to the cloud provider. We present CRIMES, an evidence based, modular security framework for cloud platforms that uses speculative execution coupled with memory introspection tools to detect malicious behavior in real time. By buffering VM outputs (i.e., outgoing network packets and disk writes) until a scan has been completed, CRIMES gives strong guarantees about the amount of damage an attack can do, while minimizing overheads. When an attack is detected, CRIMES rolls back to a recent checkpoint and performs automated forensic analysis to help pinpoint the source of an attack. Our evaluation demonstrates that CRIMES incurs less overhead compared to memory protection tools such as AddressSanitizer, while offering valuable forensic analysis for buffer overflow attacks and malware detection across multiple applications and the OS.

CCS CONCEPTS

• **Security and privacy → Virtualization and security**; *Distributed systems security*;

ACM Reference Format:
Sundaresan Rajasekaran, Harpreet Singh Chawla, Zhen Ni, Neel Shah, Emery Berger, and Timothy Wood. 2018. CRIMES: Using Evidence to Secure the Cloud. In *19th International Middleware Conference (Middleware '18), December 10–14, 2018, Rennes, France.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3274808.3274812

1 INTRODUCTION

Cloud applications are storing ever increasing volumes of data—data that is often of high value to attackers who wish to steal company secrets or personal information. Unfortunately, current cloud platforms are better prepared to offer performance management services than they are to offer security. This leaves the onus on customers to ensure the integrity of their applications.

Security remains a challenging problem for applications and operating systems. On one extreme are approaches such as virus scanners that periodically scan a system to detect evidence of attacks; this incurs relatively low overhead, but can leave a system vulnerable for long periods of times between scans. At the other end of the spectrum are memory safety tools such as Address Sanitizer [32], Safecode [14], or Intel MPX [26] which use runtimes, compilers, or specialized hardware to instrument memory accesses and detect attacks such as buffer overflows as soon as they occur. Such systems can add high cost, and only focus on a limited set of attacks within a single application. A further challenge is not only detecting attacks, but precisely pinpointing the root cause in order to prevent such an attack from happening again in the future.

In this work, we introduce a new framework to improve security of Virtual Machines (VMs) in the cloud. We call it CRIMES because it offers the following properties:

Comprehensive: Provide security against a wide set of attacks ranging from the application layer to the OS.
Responsive: Detect security threats in real time, without hurting the usability of the system.
Insightful: Automatically provide in-depth forensic analysis to help pinpoint the root cause of an attack.
Modular: Support customizable security modules to meet customer needs in diverse cloud environments.
Evidence-based: Use evidence left by attacks to efficiently detect security threats.
Safe: Detect attacks with zero-window of vulnerability so attacks have no external impact.

CRIMES achieves these characteristics with a few key techniques. First, rather than using expensive in-line checks to detect an attack (e.g., bounds checking for buffer overflows), CRIMES exploits the fact that many attacks leave *evidence* behind in memory that can be detected afterwards in a more efficient way. To prevent an attack from damaging the system between the point of an exploit and a scan, CRIMES uses *speculative execution* so that external outputs (e.g. disk writes or network packets) are only released after the integrity of the system has been verified. Finally, CRIMES uses *virtual machine introspection* (VMI) so that it can interpret the memory of a VM tofi nd evidence of attacks and automate *forensic analysis* to helpfi nd the root cause of an attack after it occurs.

Middleware '18, December 10–14, 2018, Rennes, France

ACM ISBN 978-1-4503-5702-9/18/12…$15.00
https://doi.org/10.1145/3274808.3274812

CRIMES uses an optimized continuous checkpointing mechanism that takes snapshots of VM memory as often as every 20 milliseconds to achieve speculative execution and frequent security scans. At the end of each checkpoint, it introspects into the VM's main memory for a detailed security analysis looking for evidence of attacks. If there are no anomalies found, the VM resumes speculative execution until the next security scan. On the other hand if CRIMES detects an attack, the VM is paused and a post-attack analysis is done which tries to pinpoint the root cause of the attack. Our contributions in this work are:

- A framework for online detection of attacks in VMs by leveraging memory introspection tools.
- VM checkpointing optimizations to support efficient speculative execution between security scans.
- Post-attack analysis methods that automate forensic analysis or give precise visibility at the point of an attack.

We have implemented CRIMES on the Xen virtualization platform. Our optimized checkpointing procedure improves performance by 33% compared to the Remus checkpointing system (which does not offer security guarantees) and only adds 9.8% overhead to the PARSEC benchmark suite when checkpointingfi ve times per second. CRIMES achieves this at the expense of doubling the VM's memory cost (to maintain a clean checkpoint), and using output buffering which can increase latency. For time sensitive applications such as web servers, CRIMES provides an asynchronous scanning approach that reduces overhead but weakens security guarantees. We demonstrate several potential use cases for CRIMES, including both malware analysis and buffer overflow detection. We have our code open sourced at https://github.com/SunnyRaj/crimes-4.9.

Our work on CRIMES extends our preliminary study on security as a cloud service [30]. This paper removes the requirement of a separate host for scanning services and adds multiple optimizations to reduce checkpoint overhead. Further, our complete CRIMES system supports Sync and Async scans, demonstrates malware detection and heap overflow attack prevention, and adds automated post-mortem analysis.

2 SECURITY AS A CLOUD SERVICE

Virtualization platforms have grown in popularity primarily because they facilitate application deployment and server consolidation, which has in turn led to the growth of cloud computing platforms. Today's clouds run many thousands of VMs, and offer them services such as autoscaling [23, 24, 31] and failover [13, 35]. These services can be provided at the infrastructure level, often with no modifications to customer applications.

Unfortunately, security threats in cloud environments are inevitable. Current tools that detect and prevent attacks are typically deployed by customers inside their VMs, e.g., virus scanners or memory safety compilers or runtimes. Instead, we make the case for placing security tools in the virtualization layer as a service offered by the cloud provider. This provides several benefits. First, security systems based in the hypervisor are significantly more difficult for an attacker to compromise compared to ones inside a VM [16]. Second, cloud-managed security reduces the burden on customers to keep the security software up to date in every VM. Finally, a hypervisor-based approach makes it easier to provide full system protection against threats to both applications and the operating system.

While prior work on cloud security has focused on issues such as network monitoring [34] and virus scanning [5], our work in CRIMES targets any type of attack that leaves behind some evidence observable in memory, e.g., buffer overflows or attacks that modify key kernel data structures. Tools such as libVMI [27] and Volatility [4] grant a hypervisor the ability to interpret VM memory tofi nd forensic evidence of attacks, but currently these tools are used manually for offline analysis *after an attack has happened*. The CRIMES framework allows the cloud to automatically use these in an online manner to proactively analyze each VM for signs of an attack using lightweight libVMI tools, with additional, more detailed Volatility-based forensics once an attack is found. This process is fully automated,fi tting in with the cloud computing goal of scalable, "zero-touch" management.

A key concern in providing security is the *window of vulnerability*, i.e., the length of time between when an attack occurs and when it is detected and stopped. Many recent high profile data breaches have illustrated that this time period can often stretch to months [1]. Some security systems, such as periodic virus scanners, may have windows of vulnerability of minutes or hours in order to reduce overhead by amortizing scans over a longer period. At the other extreme, memory safety techniques such as AddressSanitizer [32], SoftBound [25], and SafeCode[14] provide an ideal, zero window of vulnerability by instrumenting accesses to immediately detect attacks such as buffer overflows. This negates the impact of an attack, but it can come at high performance cost. In CRIMES, we seek to provide the best of both worlds by performing multiple scans per second and using output buffering to provide external users the same guarantee as a zero window of vulnerability approach.

Threat Model: This section outlines the threat model that CRIMES assumes and protects against. Our focus is on providing an efficient framework to analyze VMs in real time, so we assume that attacks leave evidence in memory that can be detected by one of our scanners. There are well known techniques forfi nding in-memory evidence of many attacks that CRIMES can leverage, e.g.,finding buffer overflows by checking canaries [8, 22], comparing kernel structures against known-good state to detect attacks like system call table hijacking [11], or parsing kernel data structures tofind anomalous behavior such as illicit processes [18]. We assume that an attack will leave evidence in memory, and we provide support from within the VM to ensure this happens (e.g., setting canaries in the heap). We assume that attackers are not able to erase the evidence that they leave within our scan interval, which is typically on the order of tens of milliseconds. We believe this is a reasonable assumption, because while an attack may exploit the system to gain higher privilege, it typically cannot reconstruct memory from before the attack. For example, special bits such as a heap "canary" can be created by a random number generator outside the attacker's control and inserted around objects in memory – if they are inadvertently overwritten by an attack such as a buffer overflow, the system can detect the canary is not correct. Finally, we assume an attack can affect applications and/or the OS within a VM, but that it cannot affect the system outside the VM boundary.

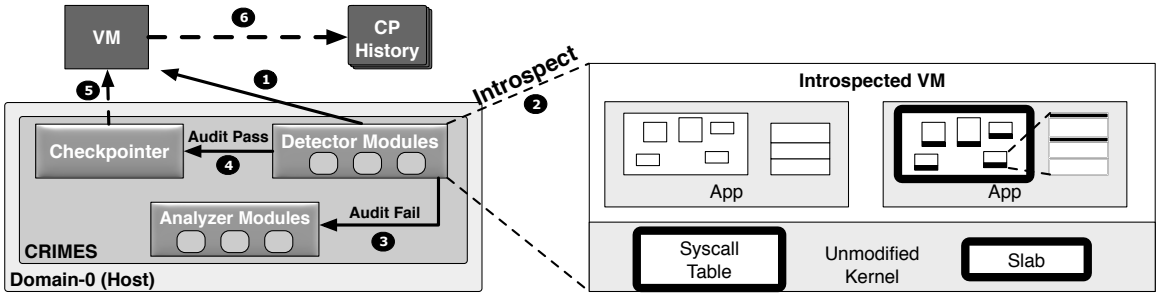


Figure 1: VMs run on cloud hosts taking checkpoints periodically for analysis. ① The Detector modulefi nds the "places" to scan, and ② uses VM introspection techniques to search for evidence of vulnerabilities, such as validating canaries in the OS and applications, or ensuring the integrity of key kernel data structures. ③ If the audit fails the Analyzer performs post-mortem analysis, else if it passes ④ Checkpointer is initiated, and ⑤ it starts taking checkpoints of the VM, andfinally ⑥ the checkpoints are stored and added to the history of previous checkpoints.

3 DESIGN

CRIMES is a modular framework for detecting attacks in the cloud. Figure 1 illustrates the design of CRIMES. Our prototype is implemented as part of the Xen hypervisor [6] and provides three key features to the VMs that run on the hypervisor. First, the *Checkpointer* speculatively executes the VM for a predetermined epoch, pauses the execution and takes a checkpoint of the VM at the end of each epoch. Second, while the VM is in a paused state, the *Detector* does a comprehensive security audit of the VM andfi nds any traces of evidence of an attack. Third, onfi nding such evidence, it stops the execution of the VM, and the *Analyzer* starts a post-attack analysis phase for a thorough forensic inspection. The rest of the section explains each of these design features in detail.

3.1 Speculative Execution

Figure 2 shows the timeline of a VM execution with CRIMES. The execution is split into epochs. During each epoch the VM is executed speculatively, i.e., the VM runs normally but with all of its external outputs such as disk writes or network packets buffered. Buffering keeps all of the VM's external outputs during the epoch in the hypervisor. At the end of each epoch, the VM is suspended, its integrity is determined by a thorough security audit, a checkpoint is created, and the buffer is released.

Buffering the VM's output guarantees that in the event of an attack during the epoch, the effect does not propagate outside of the VM. This ensures that end-users will not be affected by this attack. If the security scan at the end of the epoch succeeds, then the buffer is released, committing the result of the speculative execution period. The VM is then speculatively executed for the next epoch and the cycle continues.

In the case where the VM fails the security audit, CRIMES seeks to provide automated analysis to help identify the cause of the attack, or even reverse it. To do that, a checkpoint of the VM's state (e.g., CPU registers, memory, and disk) is created after each successful scan to provide a point that can be safely rolled back to. The VM can then be either replayed if one suspects the failure was due to a transient fault and/or an in-depth forensic analysis can be done tofi nd the root cause of the attack.

In our current implementation, CRIMES focuses on checkpointing CPU and memory state, but this can easily be extended to include disk snapshots as well [13]. Creating a checkpoint requires the VM to be paused, so this process is the source of most overhead in CRIMES—in Section 4.1 we describe several optimizations we introduce to lower this cost. Our current implementation only maintains the most recent checkpoint, however, CRIMES could be extended to include a history of checkpoints that would facilitate forensic analysis. The interval of each checkpoint epoch is a tunable parameter that is set depending on the applications that run on the VM and the level of security the VM requires. Each epoch can range from tens to a few hundred milliseconds.

A lower interval leads to more frequent audits, which reduces the impact of output buffering, but can increase checkpointing overhead. For example, a VM that hosts a web server demands quick response times (low latency), but since the network packets are buffered, the response time of a VM may be extended by the sum of the epoch interval and the time taken for the security audit to complete. In such a case of a latency sensitive VM the epoch intervals can be as low as 10 − 20ms. In the case of the VMs that aren't dependent on network or disk latency, such as CPU bound workloads, the epoch intervals can be set higher, e.g. 200ms, to reduce checkpointing overhead.

The output buffering described above can offer protection with zero window of vulnerability, thus we call this mode *Synchronous Safety*, as presented in [9]. However, not all VMs require such a strong guarantee. By disabling buffering, CRIMES can provide *Best Effort Safety*, where attacks can still be detected quickly, but they may cause some external output to leave the system before detection. However, since many current systems have windows of vulnerability of minutes or even days, Best Effort protection may still be desirable since it can offer higher performance yet only millisecond level vulnerability periods.

3.2 Detection

Determining if a VM is safe requires a robust and comprehensive audit of the VM at the end of each epoch. This section describes how CRIMES can efficiently detect and pinpoint a range of memory

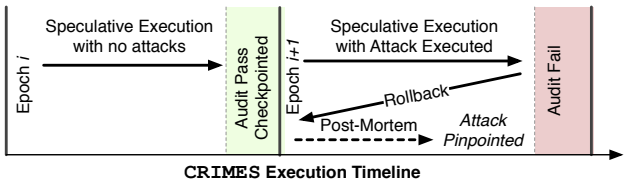


Figure 2: CRIMES speculatively executes the VM during each epoch, with comprehensive security audits at the end to verify system integrity. If an attack is detected, the VM is rolled back for a in-depth forensic analysis

errors and malware running within a VM. To interpret the inside of a VM from the hypervisor, we need information about its symbol names and the corresponding memory addresses. Virtual Machine Introspection (VMI) techniques make this possible, for example by using a System.mapfi le to locate kernel data structures for a VM running a known version of Linux [16].

VMI is mainly used for offline memory forensic analysis on VMs. VMI techniques help forensic investigators to view and analyze key user-level and kernel objects, symbols, and other key components from outside the VM. This allows a robust analysis of what happens inside the VM at any given point in time.

CRIMES can perform introspection on a VM with or without the aid of the VM. The Detector component in CRIMES provides a modular framework for different VMI-based security scans. At the end of each epoch, the Checkpointer component messages the Detector to trigger a scan.

To reduce the cost of performing audits at the end of each epoch, the Checkpointer provides the Detector with a list of dirtied pages. This allows the security audit to focus only on pages that have been modified, and thus might contain evidence of an attack. The Detector will trigger one or more of its Scan Modules, which can be customized depending on the type of attacks that a VM may face and whether the VM is made aware of the scanning system. The main goal of our work is to provide a framework to which such modules can be added to check for different types of security issues in the VM, depending on the nature of the tasks that the VM runs. Moreover, we focus on keeping the overheads of the scans minimal (within a few milliseconds).

Unaided Security Modules: In some cases it is possible to detect evidence of an attack with no assistance from within the VM. These types of scans typically look for anomalous data in well known kernel or application data structures. For example, VMI can be used to monitor the kernel's list of active tasks for blacklisted processes, or to compare the system call table against a known good state to ensure it hasn't been hijacked. Alternatively, a security module could focus on the outputs of the VM, e.g., scanning outgoing network packets for suspicious content. Such scans require basic knowledge of the VM's internals (provided by a virtual machine introspection library), but do not require any modifications inside the VM.

Guest-aided Security Modules: Many attacks do not leave evidence behind on their own, so CRIMES also supports scanning modules that get assistance from within the guest VM. This can be used to proactively install "tripwires" inside the VM's memory so that the external scan can more easilyfi nd evidence of an attack. Several prior systems have demonstrated the benefit of trip wires [9, 22]. For example, the guest could run a library to automatically place canaries after objects in the stack or heap, facilitating hypervisor-level scans for buffer overflows. This can allow CRIMES tofi nd a much broader range of attacks. We have implemented modules of each of the above types, which are detailed in Section 4.2.

3.3 Response

After a Detector module identifies a potential attack—memory errors or malware—the CRIMES Analyzer is activated to generate a comprehensive security report to aid administrators with mitigating this form of attack in the future. Before generating this report, CRIMES can optionally perform additional forensics to pinpoint the exact system state when the attack started. Once an attack is detected, CRIMES suspends the VM to prevent damage. The Analyzer's response can be broken down into two stages:

Rollback and Replay (optional): CRIMES detects attacks at the end of a checkpoint interval, but the actual exploit must have occurred sometime earlier during that epoch. To narrow down the point of the attack, CRIMES can "rollback" the VM to the checkpoint prior to the start of the attack. Then, we "replay" the VM and run it in an advanced forensics mode (determined by the Scan Module) to gatherfi ne-grained details regarding the attack. Note that during the replay phase our main goal is to determine the root cause of the attack, not provide high performance. This allows us to run more expensive security scans such as intercepting all writes to memory addresses to inspect their contents.

Postmortem Analysis: At any point, CRIMES maintains two instances of the VM: one at the current epoch (the primary VM) and another snapshot at the previous checkpoint's epoch (the backup VM). As a result, CRIMES doubles the memory requirement of the VM. If we rollback and replay the VM after detecting a potential attack, then we would have three snapshots: the two that CRIMES already maintains, and an additional at the point of the attack. If the *Detector* modulefi nds traces of an attack, then the VM is paused. At this point, we can generate two memory dumps of the VM: one at the last known safe checkpoint and the other at the point where the audit failed. Having two memory dumps around the attack significantly simplifies attack analysis. CRIMES can determine the differences between the two dumps and highlight them for an investigator.

In both cases, we use these dumps as input to the Volatility Framework. We expand on using memory dumps with Volatility for postmortem analysis in Sections 5.5 and 5.6. In our case study, we run a plethora of Volatility commands to generate a comprehensive security report of the VM.

4 IMPLEMENTATION

CRIMES's implementation leverages Xen's Remus high availability system [13] for checkpointing, and LibVMI [27] and Volatility [4] for memory analysis and forensics.

Remus speculatively executes VMs in epoch intervals by extending the final stages of Xen's live migration mechanism to take regular checkpoints of the VM's memory and disk state at the end of each interval. Remus guarantees that should the primary host go unresponsive Remus will failover to the backup VM, which then will become the primary. Another feature that Remus provides is network buffering. It buffers all the outgoing network packets of the VM in the hypervisor during an epoch interval and releases them only after receiving an acknowledgement of a complete checkpoint from the backup VM.

CRIMES builds upon Remus's framework, but uses the backup VM to represent the most recent clean snapshot, and rather than just waiting until data is sent to the backup, we perform a security audit before proceeding to the next epoch.

In this section wefi rst describe the implementation of CRIMES with three different optimizations done to Remus for providing a responsive and a comprehensive cloud security system. Then, we describe the different types of security detection and analysis modules using LibVMI library andfi nally, describe how we implement our forensic analysis tool using Volatility.

4.1 Optimizing Speculative Execution

Remus provides continuous checkpointing, but we have found that this can come at very high cost. Below we show the timeline of events at the end of each checkpoint interval for a version of Remus that has been modified to perform a VMI-based integrity scan before the checkpoint is created.

suspend Suspend active VM to paused state.
vmi Introspect the VM for integrity.
bitscan Scan bitmap to select dirty pages to propagate.
map Map VM's virtual to physical memory.
copy Propagate the VM's state to its backup.
resume Resume VM execution from suspended state.

During these steps the VM is paused, so applications can make no useful progress. In Table 1 we show the time the VM spends in each state for different intensity web workloads. In this test the backup VM runs on the same host as the primary to minimize overhead.

Workload (*ms*)	**suspend**	**vmi**	**bitscan**	**map**	**copy**	**resume**
Light	0.96	0.34	1.83	1.6	12.58	1.5
Medium	0.98	0.34	1.97	1.88	14.63	1.48
High	1.27	0.33	2.79	2.63	19.98	2

Table 1: Cost breakdown of time spent in paused state for different workloads using web-benchmark for an epoch interval of 20**ms with no optimizations**

These results show that the workload directly affects the time a VM spends in the paused state, and that this cost can rise to tens of milliseconds. Clearly this is an unacceptable cost; here the VM will run for a 20 millisecond epoch, then be paused for nearly 30 milliseconds before repeating the process in the next epoch. Copying data from the primary to backup alone takes about 70% of the total time spent in the paused state. We found that when the backup is propagated to a remote host, the overhead increased multi-fold because of the added cost of network transmission.

While the goal of Remus is to offer availability in the face of host-level failures, our focus is only on security. Instead of storing the backup on a remote machine, CRIMES keeps its checkpoints on the local host, which permits several key performance optimizations. If users desire both high availability and security, CRIMES could be configured to perform remote checkpoints and security scans. Our experiments show that this would incur minimal overhead on top of the cost of Remus. The onus is to optimize checkpointing in such a way that we spend minimum time performing tasks in the suspended state. To this end, we found several optimizations that improve the performance of our framework.

Optimization 1: `memcpy`, not `write`: There are two major factors affecting the time taken to copy data for a checkpoint. Since Remus is designed for transmitting checkpoints over the network, it uses sockets to transmit the data, which adds several unnecessary costs. First, the sockets use the `writev` system call to propagate the machine state to the backup machine. Although, `writev` is optimized over using `write` system call by grouping multiple `writes` together, writing all the machine state over the socket for each interval incurs a high overhead. Second, regardless of whether the backup was sent to a different host or the local host Remus uses `ssh` to write the memory pages over the socket. Using `ssh` encrypts the dirty page data, which is necessary for security when moving across the network, but is an unnecessary cost if the destination is the local host.

CRIMES optimizes this overhead by performing an *in-memory* copy of memory pages at each checkpoint rather than writing the pages through a socket. In the default Remus system, its Checkpointer process only maps the memory pages of the primary VM, while a "Restore" process on a remote host maps in the pages of the backup. We extend the CRIMES Checkpointer so that it maps the memory pages from both the primary VM and the backup VM into its own process address space. To acheive this, we modify the Restore code to write the backup VM's Machine Frame Numbers (MFNs) to a temporaryfi le read by the Checkpointer. Once CRIMES has the MFNs of the Backup VM it maps those addresses to its virtual address space by using the Xen's `xenforeignmemory_map` function. This allows it to use `memcpy` instead of `write` to copy modified pages from the target VM into the backup checkpoint.

Optimization 2: Global Memory Mapping: A second overhead we observed was how the Remus Checkpointer maps the pages of the primary and backup VMs for each checkpoint. In the original system, the Checkpointer maps in only the dirty pages and then unmaps them after their content has been copied. This cycle of mapping and unmapping is repeated each checkpoint interval, potentially causing high overhead. Further, the overhead is exacerbated in CRIMES, since the Checkpointer must map pages for both the primary and backup VM.

We found that the conversion of VM-specific Page Frame Numbers (PFNs) to physical Machine Frame Numbers (MFNs), and repeated mapping and unmapping each checkpoint interval incurs a very high overhead, and increases linearly with the increase in the number of dirty pages at each checkpoint. CRIMES optimizes this by maintaining a global data structure with the full PFN to MFN mapping which is loaded once at the start of the CRIMES system.

We use array as our choice of data structure because the PFNs increase from 0 to the size of the memory. Thus, all the MFNs are indexed by its corresponding PFNS making all lookups in constant $O(1)$ time. Performing the full mapping at startup increases initialization time, but eliminates the cost of repeated mappings every epoch. Since adjusting mappings requires expensive hypercalls and page table modifications, this provides a significant drop in pause time.

Optimization 3: Dirty Page Scan: Ourfi nal optimization was to modify how Remus determined which pages were dirty. Remus maintained an internal dirty bitmap data structure where each bit correspond to a page in the memory. If a particular page has been dirtied during an interval, the corresponding value in the bitmap is marked with the value 1 and 0 otherwise.

At each checkpoint, Remus searches linearly bit by bit on the dirty bitmap to determine which pages were dirtied. Although, the overhead of this method was not as significant as others, this still incurs unnecessary cost, linearly increasing with the size of the total memory.

CRIMES reduces the dirty bit scan cost based on the intuition that most pages in memory will not be dirty, and often dirty pages will be grouped consecutively. We exploit this fact by having CRIMES scan the bitmap by the machine word size (e.g., 4 or 8 bytes) since most memory regions are not modified and the bits will all be 0. Only if word is not 0, then we check each bit within that word to get the dirty pages.

4.2 Detection and Analysis Modules

We have implemented two detection and post attack analysis modules to demonstrate the types of attacks CRIMES can handle. The first targets Linux VMs and leverages a custom memory allocator in the VM to detect buffer overflow attacks, while the secondfinds illicit processes in unmodified Windows VMs and performs forensic analysis.

Malware Detection: This scanning module uses introspection to check the list of active processes running on a VM for evidence of known malware or other programs forbidden by an administrator. This mechanism needs no support from the VM, and can be done solely from the hypervisor level. The list of active processes is compared against a black-list of known malicious processes (which we obtain from [3]). If no matches are found, then the checkpointing mechanism continues. In case a match is found, the checkpointing mechanism stalls and the VM running the server pauses for further introspection in the post mortem analysis phase.

Buffer Overflow Detection: Next, we implement a canary-based buffer overflow detection [12] module. For this, we need the help of our VM to provide the scan module with the addresses of canary-protected heap objects. We create a simple malloc wrapper inside the VM for C programs. This malloc wrapper assigns a canary (or tripwire) at the end of every allocated memory object in the user space of the VM's operating system.

At the end of each epoch, the Checkpointer sends a list of dirtied pages during that checkpoint to the Detector, and the canary addresses associated with the objects allocated on these pages can then be monitored by the scanning module. The scan checks the values held by these canaries against the expected value that the canaries should hold. If the values match, that implies that the buffers associated with them haven't overflown. Otherwise, the checkpointing mechanism triggers an alert, and the VM running the server pauses for further introspection in the post-mortem analysis phase of our framework.

Once a canary is detected to have a modified value we need to find the execution path that led to the overflow. For this purpose, we need to rollback to the last checkpoint and replay the VM. This allows us to monitor the VM's memory for changes, and pinpoint the instruction which caused the buffer overflow.

We use Xen's capacity for memory event-monitoring to detect changes on the page(s) containing the corrupted canaries. The event-monitoring component leverages Xen event channels through LibVMI to monitor read, write, and execute events that occur on particular pages of memory. In Xen, each VM has a ring buffer to hold events that are to be consumed by some external service such as our forensic analysis tool. LibVMI provides abstractions to easily consume events from this ring buffer. In this security module's case, we are interested in memory events, as opposed to instructions or interrupts. LibVMI's `VMI_EVENT_MEMORY` macro enables monitoring of read/write/execute operations on a region of memory.

During the replay phase, the post mortem Analyzer will poll the events ring buffer to monitor for events to the targetted page. When an event is found, the memory operation is analyzed to see if it targets the canary (as opposed to some other portion of the page). Once the write to the canary is found the VM is paused–at the exact instruction which triggered the original overflow.

At this point, CRIMES has fully detected and replayed an attack. During that process it has created several snapshots of the VM: at the previous "good" checkpoint, at the end of the "bad" epoch, and finally at the precise point of the attack. The forensics phase of the analyzer can then utilize the Volatility Framework to provide a comprehensive security report based on these memory snapshots.

It is important to note that event monitoring with Xen is expensive. Due to its expensive nature, CRIMES does not enable event monitoring during normal operation. Instead, event monitoring is only utilized after we detect an attack and must replay it. If event monitoring was used in the former case, it would incur a significant performance loss.

Memory Forensics: In addition to the two detection and response modules described above, we have also explored several other memory forensic capabilities that can be integrated with CRIMES. CRIMES can detect hidden processes using Volatility's psscan (Windows) (which performs heuristic memory search for spotting all processes by looking into memory allocations of process stacks) and psxview (which gives a cross-section view between pslist and psscan plugins for Windows) or linux_psxview (which does a detailed analysis of processes running on a Linux machine). Therefore, any processes that appear in psscan but not in pslist for Windows or kmem_cache and pid_hash but not in pslist for Linux can potentially be malicious.

Once a malicious process has been detected, we can dump the process and get essential information, such as pid, uid, and time

PARSEC 3.0 Benchmarks	
blackscholes	Uses PDE to calculate portfolio prices
swaptions	Use HJM framework and Monte Carlo simulations
vips	Perform affine transformations and convolutions
radiosity	Compute the equilibrium distribution of light
raytrace	Simulate real-time raytracing for animations
volrend	Renders a three-dimensional volume onto a two-dimensional image plane
bodytrack	Body tracking of a person
fluidanimate	Simulate incompressiblefl uid for interactive animations
freqmine	Frequent itemset mining
water-nsquared	Solves molecular dynamics N-body problem

Table 2: Parsec Benchmarks Suite used in our experiments

stamp for deeper analysis. CRIMES can call Volatility plugins to automatically gather data such as openfi le descriptors and network connections. We talk about our Volatility memory forensic case study for buffer overflow detection on Linux machines and malware detection through signature matching on Windows machines in Section 5.5 and Section 5.6, respectively.

5 EVALUATION

5.1 Setup - machines, benchmarks

To evaluate CRIMES, we demonstrate how performance optimizations to Remus foster a speculative execution environment with which our modular security scans can run with little overhead. Our experiments are run on HP ProLiant GL160 G6 servers with two Intel Xeon *X*5650 CPUs @ 2.67 GHz and 16 GB of RAM that run Xen 4.7 with an Ubuntu 14.04, Linux kernel 3.7, Domain-0. Our VMs run OpenSUSE 13.1 with Linux kernel 4.8 running GCC version 4.8.1

We use PARSEC [7] version 3.0 for our evaluation, a common CPU and memory intensive benchmark suite. We run PARSEC benchmark with the memory safety tool AddressSanitzer [32] by compiling it with `-fsanitize=address` flag. For network workloads we chose to use `wrk`, a HTTP benchmarking tool [17] as the workload generator with NGINX version 1.11.4 as our web server.

5.2 Performance Overhead

Wefi rst evaluate the performance overhead of the CRIMES framework. All the experiments were run with both the primary and backup VM running on a single host, with network buffering unless specified otherwise. Here we use a 200ms epoch interval and evaluate the impact of epoch length in the next section.

We compare our system to Remus modified to perform a VMI scan each interval but without our optimizations (labeled **No-opt**). We also compare against Google's Address Sanitizer (**AS**), which offers protection against a variety of memory errors such as buffer overflows. We consider three variants of CRIMES:

a) **memcpy** enhances Remus with only our local memory copy optimization,
b) **Pre-map** uses both local memory copying and our preemptive mapping optimization, andfi nally,
c) **Full-opt** represents the complete CRIMES system with the above two optimization along with optimized dirty page scanning.

For simplicity, our CRIMES prototype is configured to only run a minimal no-op scan; however, we show in the next section that the scan cost is quite small since it is only performed once at the end of each interval. We expect similar performance when running more complex scanning modules.

Figure 3 shows the normalized runtime of benchmarks from the PARSEC suite under several different schemes. We report the runtime of each benchmark normalized against running it inside the same VM with no security enabled, thus a value of 2 indicates the benchmark took twice as long to complete. Using the geometric mean of all benchmarks, wefi nd that CRIMES sees an average runtime increase of only 9.8%. This is substantially better than unoptimized Remus or Address Sanitizer which increase runtime by 40 to 60%. This is because AddressSanitizer does comprehensive checks on the critical path of the application execution, while our scans are done only once per interval. We believe that adding more comprehensive checks to CRIMES will not impact the overhead significantly as the checks range between hundreds of microseconds to a few milliseconds as shown in the subsequent sections.

The runtime of CRIMES with our Full optimization is at most 50% worse than the baseline. We see the highest overhead on thefluidanimate benchmark. Here the performance of unoptimized Remus is particularly bad (nearly 5X the runtime without protection). This is because the number of memory pages dirtied byfl uidanimate for a checkpoint interval on average was 5X times higher than benchmarks with low overhead such as raytrace. CRIMES's optimized data copying provides significant improvements, particularly for benchmarks with high dirty page rates.

CRIMES consistently performs better than Address Sanitizer. In addition, our framework is able to protect against a wider range of attacks across multiple applications and the operating system. Further, CRIMES can detect some security threats with no modifications inside the VM at all.

5.3 Overhead Details

Cost Breakdown: Figure 4 shows the average time the VM running the swaptions benchmark spends in the paused state between each epoch interval over different set of optimizations.

The highest cost saving for CRIMES comes from optimizing the copy time i.e., performing an in-memory copy instead of sending the data (pages) over a socket. No-opt spends about 71% of its time in the copy phase. This includes the time it takes to initially copy the data onto a buffer and sending that data over a socket using *writev* system call to the backup VM. On the contrary, CRIMES spends only about 5% of its time for copying its data into the backup VM.

The cost of map varies for memcpy and no-opt optimization while it is constant for the other two optimizations. This is because in the case of Full and Pre-map we map the virtual addresses (PFNs)

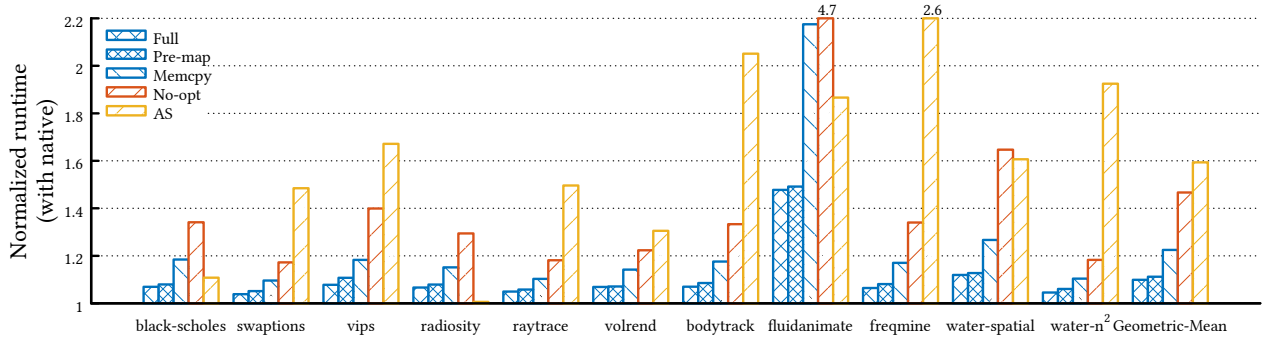


Figure 3: Normalized performance of PARSEC benchmarks with 200ms checkpoint interval

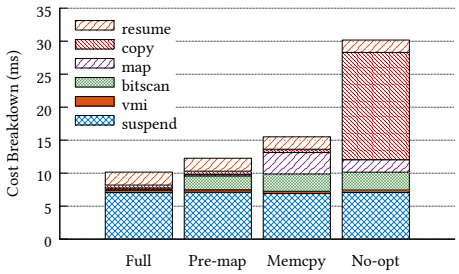


Figure 4: Absolute Cost breakdown for swaptions from PARSEC benchmark for 200ms epoch interval

to its corresponding hardware address (MFN) only once at the beginning and keep the full set of pages mapped across all intervals.

Since the Pre-map optimization is not enabled in the memcpy case, mapping takes twice as much as time as the No-opt case simply because at each epoch memcpy has to map both its primary process and the Backup VM's memory while No-opt only has to map its primary process' PFNs.

Ourfi nal piece of optimization that was included in the Full case was the cost cut from bitscan. This optimization reduces the 2.7ms time that it takes on all other optimizations down to 0.14ms for Full. Thus, with all of our optimizations combined, in the Fully Optimized case we are reducing the total pause time from 29.86ms to 10.21ms decreasing the time spent in pause state by 67%.

Interval Length: Figure 5 shows how the epoch interval affects different component of CRIMES. Figure 5b shows the normalized runtime of 4 benchmarks using Full optimization with respect to different epoch intervals. Since the PARSEC benchmark suite is very CPU and I/O intensive pausing and re-executing within very small time intervals incurs a high overhead. Thus we see that the normalized runtime decreases (performance increases) with higher the epoch interval. This is the primary reason behind using a higher checkpoint interval for any CPU and I/O intensive benchmark.

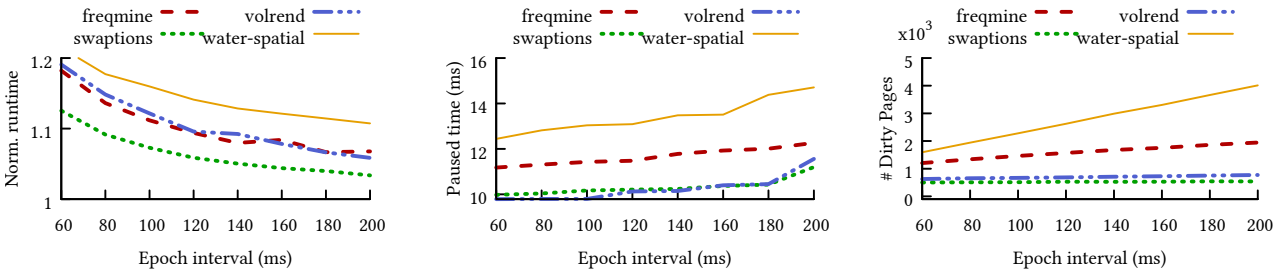
Furthermorefi gure 5b shows that with the increase in the epoch interval the pause time increases. This is directly correlated with the increase in the number of dirty pages per epoch interval shown in Figure 5c.

Figure 6a shows the normalized runtime offl uidanimate benchmark's normalized runtime with our different optimization. Regardless of the optimizations the performance of worsens with smaller epoch interval but it is imperative to see that even as the performance gets worse, with our optimizations the runtime is 3.5X faster than the No-opt case. This is a case where CRIMES performs exceptionally well because the number of dirty pages per epoch was the highest (5X) of all other benchmarks.

Bitmap Scan: To illustrate the full potential of bitmap scan optimization we simulated the cost of scanning a bitmap. We created a randomly generated bitmap representative of the size of a VM and compared the cost of scanning it bit by bit versus scanning in chunks. Figure 6b illustrates the cost of bitmap scan with respect to the size of the VM. As the VM size increases the number of pages to scan to locate the dirty pages also increases. One can see that in the bit by bit case (Not optimized) the scanning cost increases much more rapidly than the scan by chunk case (Optimized).

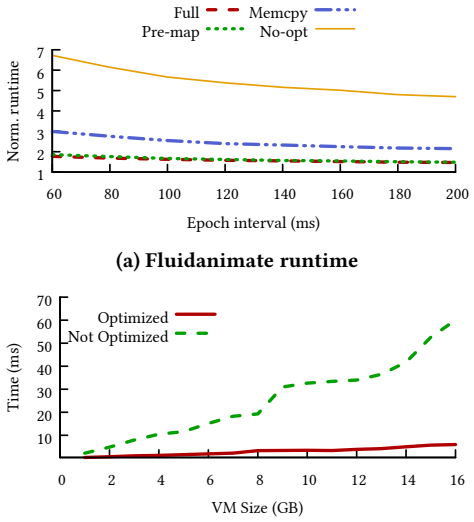
VMI Scan Cost: Invoking virtual machine introspection typically takes 100s of milliseconds to complete. However, most of this cost is related to VMI initialization, which performs routines such as detecting the operating system kernel version and configuring address translations. Since this information will not change while the VM is running, CRIMES only needs to do this once. As a result, the cost to do a VMI scan during each checkpoint can be substantially reduced. To demonstrate this, we break down the cost of running two common VMI routines: *process-list*, which walks the kernel's task queue to determine the running processes, and *module-list*, which examines the set of loaded kernel modules. We measure the time to run each of these modules 100 times on a Ubuntu Linux VM. As shown in Table 3, the preliminary LibVMI initialization consumes an average of about 66 milliseconds, with an additional 53 milliseconds used for mapping additional data structures. The actual scan of either the process list or the module list is completed in under 2 milliseconds. Only thisfi nal cost is incurred during each checkpoint in CRIMES.

Our current prototype focuses on LibVMI based introspection since it has the highest performance, with Volatility only being used for automated post-failure analysis. Volatility does not offer fundamentally different functionality than LibVMI, but its Python-based



(a) Normalized Runtime (b) Paused time (c) Dirty page per epoch

Figure 5: Normalized Performance of Parsec Benchmarks with Full Optimization



(a) Fluidanimate runtime

(b) Simulated Bitmap Scan cost

Figure 6: Optimization Benefits

Time Cost (μsec)	process-list	module-list
Initialization	67,096	66,025
Preprocessing	53,678	54,928
Memory Analysis	1,444	1,777

Table 3: LibVMI analysis costs in microseconds

interface and library of helper functions can make it easier to write complex scans. Unfortunately, Volatility has much higher cost – an average of 2.5 seconds for initialization and 500 milliseconds to perform a process scan identical to the LibVMI version. Other Volatility tools such as scanning to reveal the openfi le descriptors have similar cost. This overhead is infeasible for running synchronously at every checkpoint interval, but we note that CRIMES's maintenance of a prior checkpoint means that complex security tools such as Volatility could be used asynchronously on the last checkpoint as the VM continues to run. We leave investigation of such techniques as future work.

5.4 Impact of Best Effort Safety

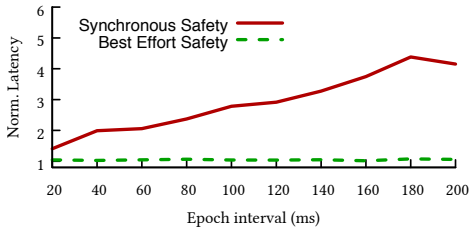
To evaluate the impact of *Best Effort Safety* we ran *NGINX* web server on the VM and measured its throughput and latency when accessing simple HTML pages. We generated the workload using the wrk benchmark as the client. Figures 7a and 7b show the normalized latency and throughput of the *wrk* benchmark with respect to the epoch interval for both *Best effort Safety* and *Synchronous safety* with Full optimization. All values shown are normalized against performance of the VM without any speculative execution, i.e., no security scans or checkpoints are being done. The average maximum throughput and latency achieved in this baseline were 17094 req/s and 2.83 ms respectively.

In the case of best-effort safety, where we do not buffer any output packets, the performance is almost equal with having no protection at all. This is because the VM is network limited, and the overhead of making checkpoints is relatively low because of the low dirty page rate.

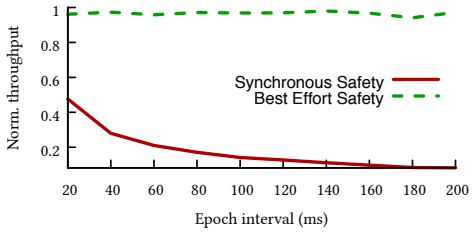
This is good, if the end-user prefers performance over a *small* compromise in security. It is hard to quantify the damage an attack can do, but we can quantify the protection in some way—the epoch interval still will determine how often we scan for attacks, so we can still guarantee that a system will be compromised for at most X milliseconds.

It is worth to note that we will still be able to detect the attack if it happened at the end of each epoch. But in case of an attack, rolling back to the previous checkpoint does not guarantee to undo the effects of the attack, especially if such an attack executes any irreversible system calls such as sending out network packets.

In contrast, with Synchronous Safety, we do buffer all outgoing packets and that impacts the performance of the VM significantly. As the interval increases the latency worsens because of buffering, and the throughput falls because packets are held for a longer period of time. This has a large effect on the three-way handshake at the start of new TCP connections. Since nginx is serving small files, this causes a high overhead. It should be noted that a larger checkpoint interval normally decreases overhead, but for the web benchmark it has the opposite effect because the incoming client workload is unable tofi ll the server to capacity because of the high latency caused by network buffering. Since our workload generator is closed loop, i.e., new connections are not created until old ones complete, the impact on throughput is exacerbated.



(a) Latency



(b) Throughput

Figure 7: Performance of Web Server

The main takeaway here is that the epoch interval and security mode should be chosen based on the workload of the VM. For instance, it is desirable to choose a large epoch interval for CPU bound VMs, while for a network intensive VM it is better to choose smaller intervals or use best effort safety. If best effort safety is sufficient, than a larger interval can be used to reduce overhead.

5.5 Case Study 1 - Overflow Attacks

In this case study we demonstrate how the CRIMES Buffer Overflow protection system can detect and respond to attacks. Inside the VM we run a simple C program built with our custom memory allocator that adds an 8 byte canary at the end of each heap object and stores a lookup table of canary addresses that can be read by the hypervisor-based scanning module. At the end of every checkpoint, the Checkpointer reports the pages dirtied during that epoch. The Detector then checks the integrity of any canaries stored within those dirtied pages. Scanning for canaries at the end of an epoch is substantially faster than instrumenting all memory accesses; we find that our scanner can validate 90,000 canaries per millisecond.

Figure 8 illustrates a timeline of the attack and response in this case study. We trigger a buffer overflow within the program, overwriting a canary, at some point within the 50 ms epoch interval at, say, time t_0 ms. After 24.4 ms, the epoch ends, the VM is paused and a scan is performed. It takes the Detector approximately 3 ms to suspend the VM and begin the scan, whichfi nishes in less than 1 ms. The canary is detected to have a modified value, thus indicating the presence of an overflow and the need for replay and forensic analysis. At time t_0 + 29 ms, CRIMESfi nishes preparing the backup checkpoint for replay and resumes its execution, starting from the checkpoint prior to the attack.

CRIMES automatically replays the VM for this epoch in order to pinpoint the exact instruction which caused the buffer overflow. The replay module accepts the address of the canary and configures

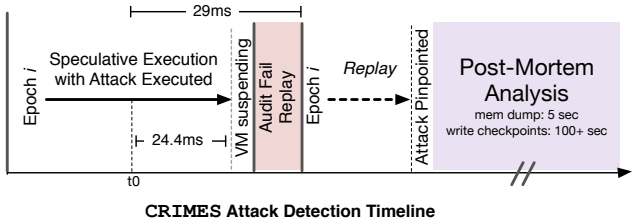


Figure 8: CRIMES detects the attack at the end of the interval and then quickly reverts to a snapshot and begins replay within 29 ms. Withinfi ve seconds CRIMES has produced a detailed forensic report, coupled with multiple full system snapshots written to disk within a few minutes.

Xen's event monitoring system to intercept any memory writes to that page by marking its protection bits.[1] By examining the address of the memory write that triggered the fault, CRIMES verifies that this is the invalid write that corrupted the canary. At this point the replay VM is suspended and post-mortem analysis is performed.

Our Buffer Overflow Post-Mortem Analysis tool uses Volatility to automatically extract useful forensics from the VM. Itfirst uses the *linux_dump_map* Volatility plug-in to extract a memory dump for the specific process and the address space, obtained using Volatility's *linux_proc_map* plug-in, that experienced the attack. This takes approximately 5 seconds on our system. This will provide valuable information for forensic analysts or developers who can look at the contents of the application's stack and heap at the instant of the attack to determine the root cause and more easily patch the application. Finally, CRIMES produces three full system checkpoints for future analysis: at the start of the epoch before the attack, at the end of the epoch after the attack, and at the point of the attack found during replay.

In total, CRIMES took under 30 ms from when the attack occurred until it completes its forensic analysis (not including the time for replay, or to save checkpoints to disk which can take tens of seconds for large VMs). The time taken for detecting an attack varies according to the frequency at which the Checkpointer takes checkpoints (in this case 50 ms). However, due to CRIMES's use of speculative execution, the outputs of the VM after the point of the exploit are all buffered, limiting external impact of the attack.

5.6 Case Study 2 - Malware Detection

Our second case study explores a windows desktop VM environment and how CRIMES can help detect unauthorized applications with no modifications to the VM itself. We create a "malware" program that reads registry information on the Windows machine, writes the data it gathers into afi le, and sends thatfi le to an external host. This represents common attacks that extract personal information from a victim's PC and transmit the data to an aggregation server.

[1] Unfortunately, we have found that the current Xen release does not fully support HVM VMs for Remus, whereas the event monitoring code only works with HVM VMs; this has prevented us from fully evaluating rollback in our system [2]. However, this is not a fundamental limitation and is expected to be resolved in upcoming Xen versions.

Our Malware Detection module checks for the presence of known malware by scanning the list of active processes against a blacklist such as McAfee's registry [3]. Of course this simple detection scheme could be extended for detection of more advanced malware threats that attempt to hide their visibility. Performing the process scanning audit at the end of each checkpoint has low overhead of approximately 0.3 microseconds. We leverage libVMI tofi nd the relevant windows kernel data structures without requiring any modifications inside the VM.

When we start our illicit program, the scanning module quickly detects it at the end of the checkpoint interval. In this case, CRIMES does not require replay of the VM since it is not looking for a specific memory event. Instead, our Malware Post Mortem Analysis is immediately triggered to automatically gather forensics about the system's behavior. The Analyzer modulefi rst uses Volatility's *procdump* plugin to extract a copy of the malware executable, which can later be analyzed in depth in a sandbox environment. Next, the Analyzer uses the *netscan* and *handles* plugins to gather forensic information about socket connections and currently openfi le handles. These are automatically performed on the checkpoints from both the start and end of the latest epoch so that they can be compared. The analysis module diffs the two outputs, producing information such as the network connection and openfi le list shown below. This provides a security analyst immediate access to useful information like the network host 104.28.18.89 that was being contacted by our "malware".

```
Malware detected:
Name            PID        Start
reg_read.exe    2656       2017-05-02 22:51:08

Open Sockets:
Protocol Local Address      Foreign Address   State
TCPv4    192.168.1.76:49164 104.28.18.89:8080 CLOSE_WAIT

Open File Handles:
\Device\HarddiskVolume2\Windows
\Device\HarddiskVolume2\Users\root\Desktop
\Device\HarddiskVolume2\Users\root\Desktop\write_file.txt
```

The Analyzer also triggers a deeper malware scan tofi nd evidence of rootkits or other threats on the system using Volatility plugins *psscan*. It also runs the *psxview* plugin which further helps locate any malware that might have hidden itself from the Windows' process structure. The results of all of this analysis is gathered into a report which can be provided to an administrator along with the system checkpoints and extracted malware executable.

6 RELATED WORK

VM Checkpointing and Speculative Execution: Continuous checkpointing of VMs was brought to Xen by Remus [13] for providing high availability by replicating VMs across a data center. SecondSite [29] was built on top of Remus to provide disaster recovery across the wide area. In addition to using checkpoints for crash failures, it has also been studied for recovering from transient errors [38], debugging [19], and performing intrusion analysis [15]. CRIMES is different from these in that it uses continuous checkpointing to provide *zero* window of vulnerability for a wide range of security scans. Additionaly, to maintain usability of the system we further optimized the Remus system to reduce its overheads.

Rollback and Replay Analysis: Several prior approaches provide deterministic record and replay techniques at different granularities. Flashback [36] allows replay and analysis for detecting software bugs, DejaView [20] allows the user to create checkpoints by combining display, operating system andfi le system virtualization without modifying applications, operating system kernels or other internal functionalities of the system, thus, allowing the user to replay and analyze the state of the system, and Crosscut [10] allows for multi-stage recording at different levels of abstraction. Crosscut comes closest to our work of performing record and replay on a virtual machine for uses such as forensic analysis. Currently CRIMES does not guarantee deterministic replay functionality, meaning that after an attack occurs it may not be possible to replay identical behavior to pinpoint the precise attack location. We believe that techniques from these prior systems could be incorporated into CRIMES to offer this functionality if necessary.

Memory Analysis: Previous approaches to memory safety such as AddressSanitizer [32], SafeCode [14], and SoftBound [25] instrument code with memory checks that run inline in an application to detect errors such as buffer overflows. These tools require special compilation with the application that needs to be monitored, and they can incur high overhead since they run within the critical path. Our work is based on ideas included in DoubleTake [22], which also uses periodic scans and speculative execution to detect memory bugs. This prior work focuses on a single application in a special runtime system, and cannot detect evidence of attacks within the operating system. CRIMES performs full system scans once every checkpoint, offering a wider range of protection modules that can be used on modified or unmodified applications, as well as the kernel. Our goal is to build a general purpose framework for security analysis. CRIMES security modules could be developed that utilize the comprehensive program instrumentation provided by a tool such as AddressSanitizer, while moving many of its checks to the periodic hypervisor-based scans to reduce overhead.

Hypervisor-based Security: We draw inspiration in our work from Aftersight [9], a hypervisor based platform for decoupling VM execution from security analysis. Aftersight relies on deterministic record and replay approaches to have a secondary VM execute the same set of instructions as a primary, while adding additional security checks. Compared to CRIMES, this requires substantially more resources (i.e., an Aftersight VM fully utilizing a CPU core will require an additional CPU core to be dedicated to the secondary, whereas CRIMES only requires additional memory). Further, Aftersight only supports single-core VMs, since enforcing system wide determinism is very expensive for multi-core setups. Similar to our work, Forensic VMs [33] rely on VMI to analyze VMs for attacks, but they cannot provide automated post attack analysis or the zero window of vulnerability guarantees offered by CRIMES. Other approaches to providing security in the hypervisor include virus scanning [28], root kit detection [39], etc. Such

systems could be incorporated into CRIMES as detection modules, granting them the ability to not only detect attacks, but perform additional analysis afterwards. Work has also been done on using virtualization to build efficient honeypot farms [37]; an extension to CRIMES would be to build a post-mortem analysis module that transforms an attacked VM into a carefully monitored honeypot to gather further information about attacks.

Virtual Machine Introspection: Virtual Machine Introspection wasfi rst introduced by Garfinkel and Rosenblum in 2003 [16]. Their work isolates the intrusion detection architecture from the monitored virtual machine, while still retaining visibility into the virtual machine's state. In the past few years, LibVMI [27] has emerged as the primary open-source introspection tool, while Volatility [4] can build on top of libVMI to offer more complex forensic operations. VMI-Honeymon [21] provides assessment and experiment on LibVMI, and successfully captures over 70 percent of 2300 malware samples.

7 CONCLUSION

Dynamic program analyses can precisely detect attacks, but their inline security checks can incur high overhead. This limits their application to a small portion of the system, e.g., certain memory accesses within a single process. Our work on CRIMES demonstrates the potential to provide full-system security services from the hypervisor layer, at relatively low cost through asynchronous security scans. From this vantage point we can use virtual machine introspection to peer inside a VM to detect evidence of a wide range of attacks across both applications and the OS. By combining introspection with frequent checkpointing, CRIMES can not only detect attacks, but roll back and replay them in order to perform automated forensic analysis. We believe this is a powerful combination that can give administrators valuable insights into the sources of an attack.

Acknowledgments: This work was supported in part by NSF grants CNS-1525992 and CNS-1525888. We thank the anonymous reviewers and our shepherd, David Eyers, for their valuable feedback on this paper.

REFERENCES

[1] 2014. Target Data Breach. http://www.ibtimes.com/timeline-targets-data-breach-aftermath-how-cybertheft-snowballed-giant-retailer-1580056 http://www.ibtimes.com/timeline-targets-data-breach-aftermath-how-cybertheft-snowballed-giant-retailer-1580056.

[2] 2017. LibVMI Events Unable to Initialize - Google Groups. https://groups.google.com/forum/#!topic/vmitools/DKzoWeLGx1c https://groups.google.com/forum/#!topic/vmitools/DKzoWeLGx1c.

[3] 2018. McAfee - View Recent Malware. https://www.mcafee.com/threat-intelligence/malware/latest.aspx https://www.mcafee.com/threat-intelligence/malware/latest.aspx.

[4] 2018. The Volatility Foundation - Open Source Memory Forensics. http://www.volatilityfoundation.org http://www.volatilityfoundation.org.

[5] Gary Anthes. 2010. Security in the cloud. *Commun. ACM* 53, 11 (2010), 16–18.

[6] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*. ACM, New York, NY, USA, 164–177. https://doi.org/10.1145/945445.945462

[7] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors.* Ph.D. Dissertation. Princeton University.

[8] Gang Chen, Hai Jin, Deqing Zou, Bing Bing Zhou, Zhenkai Liang, Weide Zheng, and Xuanhua Shi. 2013. Safestack: Automatically patching stack-based buffer overflow vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* 10, 6 (2013), 368–379.

[9] Jim Chow, Tal Garfinkel, and Peter M. Chen. 2008. Decoupling Dynamic Program Analysis from Execution in Virtual Environments. In *USENIX 2008 Annual Technical Conference (ATC'08)*. USENIX Association, Berkeley, CA, USA, 1–14. http://dl.acm.org/citation.cfm?id=1404014.1404015

[10] Jim Chow, Dominic Lucchetti, Tal Garfinkel, Geoffrey Lefebvre, Ryan Gardner, Joshua Mason, Sam Small, and Peter M. Chen. 2010. Multi-stage Replay with Crosscut. *SIGPLAN Not.* 45, 7 (March 2010), 13–24. https://doi.org/10.1145/1837854.1736002

[11] Mihai Christodorescu, Reiner Sailer, Douglas Lee Schales, Daniele Sgandurra, and Diego Zamboni. 2009. Cloud security is not (just) virtualization security: a short paper. In *Proceedings of the 2009 ACM workshop on Cloud computing security*. ACM, 97–102.

[12] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. 1998. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7 (SSYM'98)*. USENIX Association, Berkeley, CA, USA, 5–5. http://dl.acm.org/citation.cfm?id=1267549.1267554

[13] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. 2008. Remus: High Availability via Asynchronous Virtual Machine Replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)*. USENIX Association, Berkeley, CA, USA, 161–174. http://dl.acm.org/citation.cfm?id=1387589.1387601

[14] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. 2006. SAFECode: Enforcing Alias Analysis for Weakly Typed Languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, NY, USA, 144–157. https://doi.org/10.1145/1133981.1133999

[15] George W Dunlap, Samuel T King, Sukru Cinar, Murtaza A Basrai, and Peter M Chen. 2002. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review* 36, SI (2002), 211–224.

[16] Tal Garfinkel, Mendel Rosenblum, and others. 2003. A Virtual Machine Introspection Based Architecture for Intrusion Detection.. In *Networked and Distributed System Security Symposium (NDSS)*, Vol. 3. 191–206.

[17] Will Glozer. 2018. wrk - a HTTP benchmarking tool. https://github.com/wg/wrk https://github.com/wg/wrk.

[18] Amarnath Jasti, Payal Shah, Rajeev Nagaraj, and Ravi Pendse. 2010. Security in multi-tenancy cloud. In *Security Technology (ICCST), 2010 IEEE International Carnahan Conference on*. IEEE, 35–41.

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

[20] Oren Laadan, Ricardo A. Baratto, Dan B. Phung, Shaya Potter, and Jason Nieh. 2007. DejaView: A Personal Virtual Computer Recorder. *SIGOPS Oper. Syst. Rev.* 41, 6 (Oct. 2007), 279–292. https://doi.org/10.1145/1323293.1294289

[21] Tamas K. Lengyel, Justin Neumann, Steve Maresca, and Bryan Payne. [n. d.]. Virtual Machine Introspection in a Hybrid Honeypot Architecture. In *Presented as part of the 5th Workshop on Cyber Security Experimentation and Test*. USENIX, Bellevue, WA. https://www.usenix.org/conference/cset12/workshop-program/presentation/Lengyel

[22] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2016. DoubleTake: Fast and Precise Error Detection via Evidence-Based Dynamic Analysis. In *International Conference on Software Engineering*. http://lib-arxiv-008.serverfarm.cornell.edu/abs/1601.07962

[23] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A Lozano. 2014. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing* 12, 4 (2014), 559–592.

[24] Ming Mao, Jie Li, and Marty Humphrey. 2010. Cloud auto-scaling with deadline and budget constraints. In *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on*. IEEE, 41–48.

[25] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 245–258. https://doi.org/10.1145/1542476.1542504

[26] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2017. Intel MPX Explained: An Empirical Study of Intel MPX and Software-based Bounds Checking Approaches. *CoRR* abs/1702.00719 (2017). http://arxiv.org/abs/1702.00719

[27] Bryan D Payne. 2012. Simplifying virtual machine introspection using libvmi. *Sandia report* (2012).

[28] Daniel Quist, Lorie Liebrock, and Joshua Neil. 2011. Improving antivirus accuracy with hypervisor assisted analysis. *Journal in Computer Virology* 7, 2 (May 2011), 121–131. https://doi.org/10.1007/s11416-010-0142-4

[29] Shriram Rajagopalan, Brendan Cully, Ryan O'Connor, and Andrew Warfield. 2012. SecondSite: Disaster Tolerance As a Service. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE '12)*. ACM, New York, NY, USA, 97–108. https://doi.org/10.1145/2151024.2151039

[30] Sundaresan Rajasekaran, Zhen Ni, Harpreet Singh Chawla, Neel Shah, Timothy Wood, and Emery Berger. 2016. Scalable Cloud Security via Asynchronous Virtual Machine Introspection. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. USENIX Association, Denver, CO. https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/rajasekaran

[31] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. 2011. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*. IEEE, 500–507.

[32] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC'12)*. USENIX Association, Berkeley, CA, USA, 28–28. http://dl.acm.org/citation.cfm?id=2342821.2342849

[33] A. L. Shaw, B. Bordbar, J. Saxon, K. Harrison, and C. I. Dalton. 2014. Forensic Virtual Machines: Dynamic Defence in the Cloud via Introspection. In *2014 IEEE International Conference on Cloud Engineering*. 303–310. https://doi.org/10.1109/IC2E.2014.59

[34] Seungwon Shin and Guofei Gu. 2012. CloudWatcher: Network security monitoring using OpenFlow in dynamic cloud networks (or: How to provide security monitoring as a service in clouds?). In *Network Protocols (ICNP), 2012 20th IEEE International Conference on*. IEEE, 1–6.

[35] Dilbag Singh, Jaswinder Singh, and Amit Chhabra. 2012. High availability of clouds: Failover strategies for cloud computing using integrated checkpointing algorithms. In *Communication Systems and Network Technologies (CSNT), 2012 International Conference on*. IEEE, 698–703.

[36] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. 2004. Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '04)*. USENIX Association, Berkeley, CA, USA, 3–3. http://dl.acm.org/citation.cfm?id=1247415.1247418

[37] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. 2005. Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP '05)*. ACM, New York, NY, USA, 148–162. https://doi.org/10.1145/1095810.1095825

[38] Long Wang, Zbigniew Kalbarczyk, Ravishankar K Iyer, and Arun Iyengar. 2010. Checkpointing virtual machines against transient errors. In *On-Line Testing Symposium (IOLTS), 2010 IEEE 16th International*. IEEE, 97–102.

[39] Xiongwei Xie and Weichao Wang. 2013. Rootkit detection on virtual machines through deep information extraction at hypervisor-level. In *2013 IEEE Conference on Communications and Network Security (CNS)*. 498–503. https://doi.org/10.1109/CNS.2013.6682767