Trail of Bytes: Efficient Support for Forensic Analysis

Srinivas Krishnan
Dept. of Computer Science
University of North Carolina at Chapel Hill
krishnan@cs.unc.edu

Kevin Snow
Dept. of Computer Science
University of North Carolina at Chapel Hill
kzsnow@cs.unc.edu

Fabian Monrose
Dept. of Computer Science
University of North Carolina at Chapel Hill
fabian@cs.unc.edu

ABSTRACT

For the most part, forensic analysis of computer systems requires that one first identify suspicious objects or events, and then examine them in enough detail to form a hypothesis as to their cause and effect [34]. Sadly, while our ability to gather vast amounts of data has improved significantly over the past two decades, it is all too often the case that we tend to lack detailed information just when we need it the most. Simply put, the current state of computer forensics leaves much to be desired. In this paper, we attempt to improve on the state of the art by providing a forensic platform that transparently monitors and records data access events within a virtualized environment using only the abstractions exposed by the hypervisor. Our approach monitors accesses to objects on disk and follows the causal chain of these accesses across processes, even after the objects are copied into memory. Our forensic layer records these transactions in a version-based audit log that allows for faithful, and efficient, reconstruction of the recorded events and the changes they induced. To demonstrate the utility of our approach, we provide an extensive empirical evaluation, including a real-world case study demonstrating how our platform can be used to reconstruct valuable information about the *what*, *when*, and *how*, after a compromised has been detected.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Information flow, invasive software, security kernels; K.6.5 [**Management of Computing and Information Systems**]: Unauthorized access, logging and recovery

General Terms

Experimentation, Security

Keywords

Forensics, Virtualization, Provenance, Audit

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CCS'10, October 4–8, 2010, Chicago, Illinois, USA.
Copyright 2010 ACM 978-1-4503-0244-9/10/10 ...$10.00.

1. INTRODUCTION

Today, postmortem intrusion analysis is an all too familiar problem. Our devices are repeatedly compromised while performing seemingly benign activities like browsing the Web [32], interacting on social-networking websites, or by malicious actors that use botnets as platforms for various nefarious activities [10]. Sometimes, the threats can also arise from the inside (e.g., corporate espionage), and often lead to substantial financial losses. Underscoring each of these security breaches is the need to reconstruct past events to know *what* happened and to better understand *how* a particular compromise may have occurred. Sadly, although there has been significant improvements in computer systems over the last few decades; data forensics remains a very tedious process; partly because the detailed information we require to reliably reconstruct events is simply not there, when we need it the most [9].

Loosely speaking, recent efforts in data forensic research have focused on tracking changes to file system objects by using monitoring code resident in kernel space, or by making changes to the application binary interface. However, without proper isolation these approaches are subject to tampering and therefore can not provide strong guarantees with respect to the integrity of the recorded events. Malicious users can, for instance, inject code into either kernel or user space, thereby undermining the integrity of the logs maintained by the tracking mechanism. Virtualization [15] provides a potential avenue for enabling the prerequisite isolation criteria by providing a sandbox for operating system code and applications. For example, a hypervisor can mediate disk accesses at the block level by presenting a virtual disk to the virtual machine (VM). An obvious disadvantage, however, is that this abstraction suffers from a "semantic gap" problem [3], in which the mapping between file-system objects and disk blocks are lost, thereby making it difficult to track objects beyond the disk layer.

In this paper, we propose an approach for monitoring accesses to data in a virtualized environment while bridging the semantic gap issue. Specifically, we provide an approach for monitoring accesses to data that originated from disk, and capture subsequent accesses to that data in memory—even across different processes. Our approach achieves this goal without any monitoring code resident in the virtual machine, and operates purely on the abstractions provided by the hypervisor. Operating at this layer mandates that we access the disk at the block layer, memory at the physical frame layer and system calls at the instruction layer—all of which offer substantial engineering challenges of their own.

In that regard, our main contributions are in the design and implementation of an accurate monitoring and reconstruction mechanism that collates and stores events collected at different levels of abstraction. We also provide a rich query interface for mining the captured information. This provides the forensic analyst with detailed information to aide them in understanding what transpired after a compromise (be it a suspicious transfer of data or modification of files) has been detected. We also provide an extensive empirical analysis of our platform, including a real world case study using our framework.

The remainder of the paper is organized as follows. We first present some background and related work in Section 2. Sections 3 and 4 describes our design and architecture, including the various monitoring subsystems and the respective challenges in combining data from the various levels of abstraction. In Section 5, we present a detailed empirical evaluation of the runtime overheads and accuracy of our logging and reconstruction techniques. To highlight the strength of our approach even further, we present a case study in Section 6 showing how the framework was used to uncover interesting forensic evidence from a laptop that had been connected to a public network for one week. We discuss attacks on, and limitations of, our current design in Section 7 and conclude in Section 8.

2. BACKGROUND AND RELATED WORK

Generally speaking, computer forensics attempts to answer the question of who, what and how after a security breach has occurred [34]. The fidelity of the recorded information used in such analyses is highly dependent on how the data was collected in the first place. Keeping this in mind, the approaches explored in the literature to date can be broadly classified as either client-based approaches (that use application or kernel-based logging) or virtualization-based approaches (that use hypervisor based logging). While client-based approaches can provide semantic-rich information to a security analyst, their fidelity can be easily undermined as the logging framework is usually resident within the same system that it is monitoring. Hypervisor-based approaches, on the other hand, are generally thought to lack the semantic detail of client-based approaches, but can achieve greater resistance to tampering as the logging mechanisms reside in privileged sandboxes outside the monitored system.

Client-based Approaches.

File-system integrity and verification has a long history, with some early notable examples being the work of Spafford *et al.* on Tripwire [21] and Vincenzetti *et al.* on ATP [37]; both of which use integrity checks to verify system binaries (e.g., `/sbin/login`). Extending this idea further, Taser [14] detects unauthorized changes to the file-system and reverts to a known good state once malfeasance is detected. Solitude [16] extends this concept even further by using a copy-on-write solution to selectively rollback files, thereby limiting the amount of user data that would be lost by completely reverting to the last known good state. These systems do not record evidence on how an attack occurred and the data that was compromised instead they are geared primarily at efficient restoration back to a known good state.

More germane to our goals are systems such as PASS [28] and derivatives thereof (e.g., [29]) that provide data provenance by maintaining meta-data in the guest via modifications to the file-system. However, this requires extensive guest modifications and shares the same problems of client-based systems.

Virtualization-Based Approaches.

In order for virtualization-based approaches to work in a data forensic framework, they need to first overcome the disconnect in semantic views at different layers in an operating system [3, 12]. In particular, Chen *et al.* [3] provides excellent insight into advantages and disadvantages of implementing secure systems at the hypervisor layer. The challenges are generally related to performance and the difference in abstractions between the hypervisor layer and the guest virtual machine. While the issue of performance has been addressed as hypervisor technologies mature, the "semantic gap" still remains. Antfarm [19], Geiger [20] and VMWatcher [18], have successfully bridged this gap for a given layer of abstraction, but to the best of our knowledge, no single work has tackled the problem of bridging the gap for a set of interconnected layers of abstraction (i.e., spanning disk, memory and processes) while preserving the causal chain of data movement.

Closely related in goals is the approach of King *et al.* [22] which provides an event reconstruction approach for relating processes and files. BackTracker reconstructs events over time by using a modified Linux kernel to log system calls and relate those calls based on OS-level objects [23]. The semantic gap issue is bridged by parsing the memory contents of the virtual machine during the introspection time using a EventLogger compiled with the virtual machine's kernel headers. This approach is fragile as any changes to the guest kernel will undermine their approach [23, 22]. Similarly, in their VM-based approach, it is not possible to monitor operating systems that are closed-source. While BackTracker made significant strides in this area, we find that relying on just system calls to glean OS state is not enough for a number of reasons. For instance, since it does not monitor memory events, data movements (such as a process sending a file over a network socket) can only be inferred as "potential" causal relationships; neither can it detect the exact object that was sent over the network. To be fair, these were not part of its stated goals. By contrast, the causal relationships we build attempts to capture access chains across processes, all-the-while storing the exact content that was accessed and/or modified.

Also relevant are the techniques used by Patagonix [27] and XenAccess [31] that employ forms of memory inspection for VM introspection. Patagonix's goal is to detect changes between binaries on disk and their image in memory. XenAcess is positioned as an extensible platform for VM monitoring. Our goals and approach is different in that we use signals from different layers of the VM (i.e., the system-call, memory and storage layers) to correlate accesses to a monitored object. Lastly, this work significantly extends our preliminary work [24].

3. DATA TRACKING

Our primary goal in this paper is to enable fast and efficient recording of events involving a monitored data store (e.g., a disk partition), at a granularity that allows a security analyst to quickly reconstruct detailed information about accesses to objects at that location. Conceptually, our approach is composed of two parts, namely an efficient

monitoring and logging framework, and a rich query system for supporting operations on the recorded data. To support our goals, we monitor events to a collection of locations L (i.e., memory, disk or network) and record read or write operations on L. We denote these operations as O. Any additional operations (e.g., create or delete) can be modeled as a combination of these base operations. We tie these accesses to the corresponding causal entity that made them, to ensure that a forensic analyst has meaningful semantic information for their exploration [2].

The approach we take to capture these causal relationships is based on an event-based model, where events are defined as accesses, O, on a location L caused by a some entity, i.e., $E_i(O, L) \rightarrow ID$. Loosely speaking, an entity is modeled as the set of code pages resident in a process' address space during an event. The distinct set of code pages belonging to that process is then mapped to a unique identifier. This event-based model also allows us to automatically record events that are causally related to each other, and to chain the sequences of events as $\bigcup_i^n E_i$. Intuitively, events are causally related based on the same data being accessed from multiple locations; i.e., we consider $E_0(O, L)$ to be causally related to $E_1(O', L')$ if the same data object resides in L and L'.

Since the hypervisor views the internals of a virtual machine as a black box, a key challenge is in realizing this model with minimal loss of semantic information. This challenge stems from the fact that the monitoring subsystem gets disjoint views of operational semantics at different levels of abstraction. For example, a read system call operates with parameters in virtual memory and the guest file system layer, which then spawns kernel threads to translate the file system parameters into blocks; leading to the request finally being placed on the I/O queue. Without any code in the guest, the challenge is in translating these requests and chaining them together as a single event.

As we show later, one contribution of this work lies in our ability to link together the various events captured within the hypervisor. In what follows, we present our architecture and the design choices we made in building a platform that realizes the aforementioned model.

3.1 Architecture

The monitoring framework is built on top of Xen [1] with hardware-virtualization [26]. At a high level, the Xen hypervisor is composed of a privileged domain and a virtual machine monitor (VMM). The privileged domain is used to provide device support to the unprivileged guests via emulated devices. The VMM, on the other hand, manages the physical CPU and memory and provides the guest with a virtualized view of the system resources. This allows the monitoring framework to monitor—from the hypervisor—specific events that occur in the virtual machine.

The framework is composed of three modules that monitor disk, memory, and system calls (see Figure 1). The modules are fully contained within the hypervisor with no code resident in the virtual machine. The system is initiated by monitoring accesses to a specific set of virtual machine disk blocks on the virtual disk. The storage module monitors all direct accesses to these blocks and their corresponding objects, while subsequent accesses to these objects are tracked via the memory and system call modules. Specifically, the memory module in conjunction with the system call module allows the framework to monitor accesses to the object after it has been paged-in to memory, and also builds causal relationships between accesses. The memory module is also responsible for implementing the mapping function that allows us to tie events to specific processes.

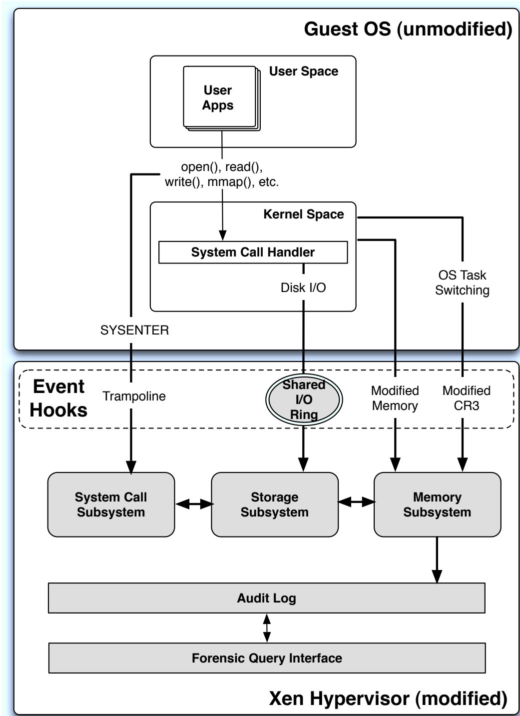

Figure 1: Overall architecture of the forensic platform, depicting the memory, storage and system call layers

As a result of our design, each of these modules have to bridge the "semantic gap" prevalent at that layer of abstraction; i.e., blocks to files, machine physical addresses to guest virtual addresses, and instructions to system calls. Since the framework is built to log events happening in the guest, a single guest event might trigger multiple hypervisor events crossing various abstraction boundaries, e.g., consecutive writes to a file by a text editor will require disk objects to be mapped back to the file, writes to the page in the guest's memory has to be mapped to the actual page in physical memory, etc. To effectively observe these linkages, our modules work in tandem using a novel set of heuristics to link events together. These events are stored in a version-based audit log, which contains timestamped sequences of reads and writes, along with the corresponding code pages that induced these changes. We now turn our attention to the specific functionality of each of the monitoring modules.

3.1.1 Storage Subsystem

The storage module is the initialization point for the entire monitoring framework. That is, a specific range of virtual machine disk blocks are monitored via a watchlist maintained by this module. Any disk accesses to the objects on the watchlist triggers updates to the storage module. The accesses to blocks on the watchlist also notifies the memory module to monitor the physical page where the blocks are paged-in. In what follows, we first discuss how we monitor access at the block layer.

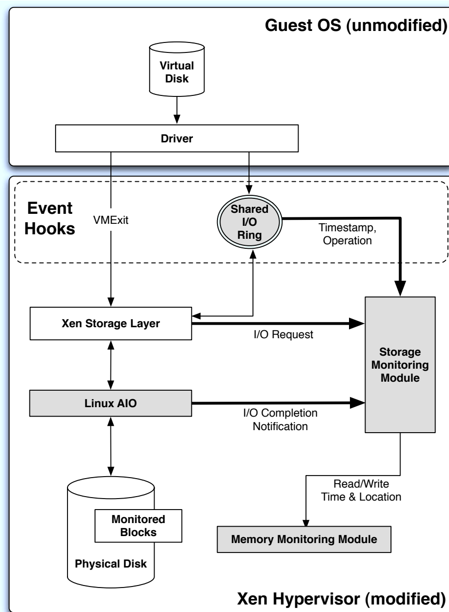

Figure 2: Overview of the storage monitoring module, showing our hooks for monitoring disk I/O at the Xen Storage and Linux AIO layers

Figure 2 describes the Xen storage model and the enhancements we made to monitor disk I/O. In Xen, block devices are supported via the Virtual Block Device layer. Guests running on top of Xen see a virtual hard disk and therefore cannot directly modify physical disk blocks. Specifically, all accesses are mediated through the Xen storage layer, which exposes an emulated virtual disk. All I/O requests from the guest are written to an I/O ring, and are consumed by the Xen storage layer.

The storage module monitors the physical blocks on this virtual disk and automatically adds them to watchlist it maintains. As guests place their I/O requests onto the shared ring, our monitoring code is notified via a callback mechanism of any accesses to the blocks on the watchlist. This allows us to timestamp a request as soon as it hits the I/O ring—which is critical in matching the access with the syscall that made the request, enabling the memory module to link a disk access with a specific process. Finally, the storage module waits for all reads/writes to be *completed* from disk before committing an entry in our logging data-structure.

As alluded to above, accesses to disk blocks typically happen as the result of a system call. In order to tie these two events together, it is imperative that we also monitor events at the system call layer. Next, we examine how we achieve this goal.

3.1.2 System Call Monitoring Subsystem

The system call module is responsible for determining when the guest makes system calls to locations of interest ($L =$ disk, memory or network), parsing the calls and building semantic linkage between related calls. First, we describe how the module monitors the system calls and then discuss how they are used to build semantic linkages in conjunction with the memory monitoring module.

Monitoring System Calls

The use of hardware virtualization makes the efficient tracking of system calls in the guest an interesting challenge. To see why, notice that system calls on the x86 platform can be made by issuing either a soft interrupt `0x80` or by using fast syscalls (i.e., `SYSENTER`). Modern operating systems use the latter as it is more efficient. This optimized case introduces an interesting challenge: a traditional `0x80` would force a `VMExit` (thereby allowing one to trap the call), but fast syscalls on modern hardware virtualized platforms ***do not induce a*** `VMExit`. However, syscalls must still retrieve the target entry point (in the VM's kernel) by examining a well-known machine specific register (`MSR`)[1]. Similar approaches for notification on system call events at the hypervisor layer have also been used recently in platforms like Ether [7].

Since the hypervisor sets up the `MSR` locations, it can monitor accesses to them. Our solution involves modifying the hypervisor to load a trampoline function (instead of the kernel target entry) on access to the `MSR` for syscalls. The trampoline consists of about 8 lines of assembly code that simply reads the value in `eax`[2] and checks if we are interested in monitoring that particular system call before jumping into the kernel target point. If we are, then the memory module (Section 3.1.3) is triggered to check the parameters of the call to see if they are accessing objects on the memory module's watchlist. The trampoline code runs inline with virtual machine's execution and does not require a trap to the hypervisor, avoiding the costly `VMEXIT`.

Capturing the Semantic Linkage

The system call module in conjunction with the memory module is responsible for building the semantic linkage between a set of related calls, for example, a `read()` call on a file whose blocks we monitor and a subsequent socket `open(),write()` of the bytes to a network socket. In order to achieve this goal we selectively monitor types of syscalls that could yield operations in our event model.

Specifically, we monitor syscalls that can be broadly classified as involving (1) *file system* objects, e.g., file open, read, write (2) *memory resident* objects, e.g., mmap operations (3) *shared memory* objects, e.g., ipc, pipes and (4) *network* objects, e.g., socket open and writes. As described earlier the system call module will monitor these calls and parse the parameters. The approach we then take to create linkages between such calls is straightforward: we simply examine the source and destination parameters to infer data movement. In this example, the system call monitor will be triggered on each of the file `read()`, network socket `open()` and `write()` calls. Since the source parameter of the `read()` references a monitored page, the memory module notifies the system call module of the offending access, and also adds the corresponding page of the destination parameter (e.g., the buffer) to its watchlist. When the memory module is later triggered because of the write on a network socket, that access will also be returned as an "offending" access since it references a page that is now on the memory module's watchlist. As a result, the system call module will connect the two calls and build the semantic linkage. Unlike other approaches that attempt to infer causal linkages based on data movements, our platform is able to accurately and definitively link events that are causally related. We now discuss the specifics of how the

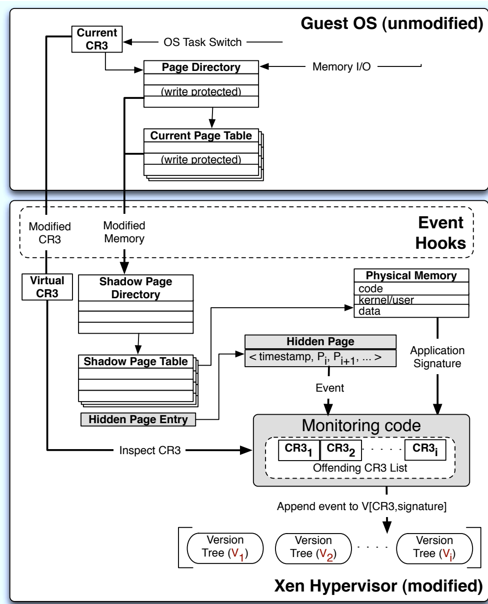

Figure 3: Overview of the memory monitoring module, showing the hooks needed for tracking of monitored objects in memory and for logging the offending processes.

memory module decides if a particular event is accessing a monitored object.

3.1.3 Memory Monitoring Subsystem

The key function of this module is to track accesses to monitored objects once they are *resident* in memory. Recall that the initial access to L on disk causes the storage module to notify the memory module of potential data movement. This access causes a page fault, as the object has not yet been paged into memory. Since Xen manages the physical memory and hardware page tables, the fault is handled by the hypervisor. Our memory monitoring module is notified of this fault via the callback placed in Xen's shadow page table mechanism, and updates its watchlist with the machine physical page of the newly paged-in monitored object. For brevity sakes, we omit system level details and provide only the essential details. Before we proceed, we simply note that Xen provides the VM with a virtualized view of the physical memory by performing the actual translation from guest physical pages to actual machine physical pages. Further details can be found in [1].

Tracking objects

The memory module uses its watchlist to track all subsequent accesses to monitored objects in memory. Recall that the system call module consults the memory module to determine if an access is to a protected object. To make this determination, the memory module consults its watchlist, and returns the result to the system call module.[3]

Notice that the memory monitoring module is in no way restricted to tracking only events triggered via system calls. Since it monitors objects in physical memory, *any* direct accesses to the object will be tracked. For instance, accesses to objects in the operating systems buffer cache will always trigger a check of the memory module's watchlist.

Our approach extends the coverage of events even to accesses that might occur on monitored objects that are copied over to other memory locations. Since the memory monitoring module is triggered from the initial page-in event of the monitored data block from disk into memory, this paged-in machine physical page is automatically added to the watchlist. Hence, any subsequent events on this page such as a `memcpy`() will result in the target memory location of the copy operation to be also added to the watchlist[4]. This is done to prevent evasion techniques that might copy the data into a buffer and then send the data over a network socket. Hence, any indirect data exfiltration attempts will also be recorded as an access to the original monitored block.

This is a key difference between the type of taint tracking [6, 4] commonly used to track objects in memory and the physical page monitoring we propose. Although taint tracking of that type affords for monitoring accesses to memory locations at a very fine granularity (e.g. pointer tracking), it does incur high overhead [36]. The memory tracking we implemented tracks accesses to the initial physical page frame where the data from monitored storage was paged in and subsequent physical memory locations the data was copied to. Our low overhead is achieved via a copy-on-write mechanism that tracks subsequent changes and accesses to the monitored objects. This implementation affords a coarser mechanism compared to taint tracking for memory monitoring, but achieves our goals at a much lower cost.

Once the decision is made that an access is to a monitored object, the memory module notes this event by timestamping the access [5]. The module also stores a "signature" of the code pages of the offending process. Recall that the `CR3` register on the x86 platform points to the page directory of the *currently executing* process within the VM. Hence, to keep our overheads low, we do the signature creation lazily and add the address of the `CR3` register (page-table register) to a queue of offending addresses that must be extracted later.

The signature is created as follows. For each item on this queue, we examine its page frames to inspect those codepages that are unique to the process being inspected. Since a `CR3` could potentially point to different processes over time, we log the accesses in a modified B+–tree [33] where the root node is indexed by the tuple $\langle$`CR3`, set of codepages$\rangle$. In this way, we avert appending a new process' events to an old process' log. We call this structure a version-tree. The keys to the version-tree are the block numbers corresponding to the monitored object on disk, and the leaves are append-only entries of recorded operations on location L. The version-tree is built as follows:

1. If no version-tree exists for the process we are examining i.e. no tree has a root node that equals the current `CR3` and code page hash, then let the set of known codepages be $S = \emptyset$, and skip to step (3).

2. Compare the hash of the codepages in the page table to the stored value in the tree. If the hashes are the same, there are no new codepages to record, and we only need to update the accesses made by this process; therefore, proceed to step (4).

3. To determine what new codepages have been loaded into memory, compute the cryptographic hash of the contents of the individual pages, c_i. Next, for each $h(c_i) \notin S$, determine whether it is a `kernel` or `user` page (e.g., based on the U/S bit), and label the page

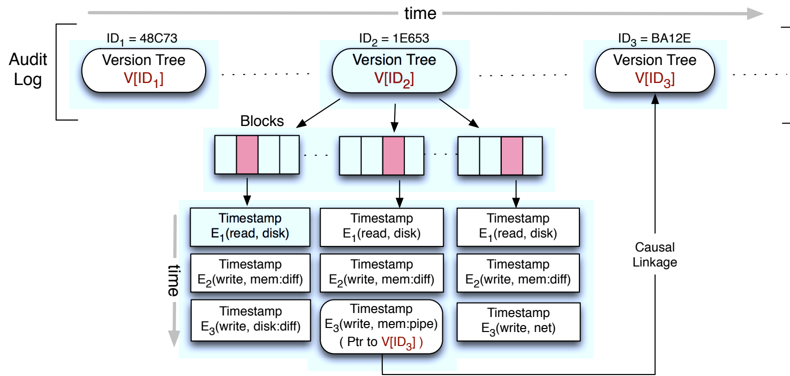

Figure 4: The version tree stores different versions of blocks and the corresponding codepages that accessed these blocks over time. To support efficient processing of the audit log, we also store pointers to other version-trees of causally related processes.

accordingly. If $h(c_i)$ is found in page tables of more than one process, then label that page as `shared`.

4. Let S' be the set containing the hashes of `user` pages. Insert the access patterns (i.e., $E_0(O, L), \ldots, E_1(O, L)$) into the version-tree with root node $\langle \texttt{CR3}, S \rangle$. That is, store the access time, location L, and "diffs" of the changed blocks for write operations, into the version-tree for that process. Update the root node to be the tuple $\langle \texttt{CR3}, S \cup S' \rangle$.

These version-trees are periodically written to disk and stored as an audit log where each record in the log is itself a version-tree (see Figure 4). Whenever the system call module notes a casual relationship between entities accessing the same monitored objects—e.g., $E_i(O, L)$ by entity p_1 and $E_j(O', L')$ by p_2—we add a pointer in the version tree of p_1 to p_2. These pointers help with efficient processing of the audit log. Having recorded the accesses to objects in L, we now discuss how the logs can be mined to reconstruct detailed information to aid in forensic discovery.

4. MINING THE AUDIT LOG

To enable efficient processing of the data during forensic analysis, we support several built-in operations in our current prototype. These operators form our base operations, but can be combined to further explore the audit log. For the analyses we show later, the operations below were sufficient to recover detailed information after a system compromise.

- `report`$(w, \mathcal{B})$: searches all the version trees and returns a list of IDs and corresponding accesses to any block $b \in \mathcal{B}$ during time window w.

- `report`(w, ID): returns all blocks accessed by ID during time window w.

- `report`$(w,$ *access*$, \mathcal{B} \mid ID)$: returns all operations of type *access* on any block $b \in \mathcal{B}$, or by ID, during time window w.

- `report`$(w,$ `causal`$, \mathcal{B} \mid ID)$: returns a sequence of events that are causally related based on either access to blocks $b \in \mathcal{B}$, or by ID, during time window w.

4.1 Mapping blocks to files

Obviously, individual blocks by themselves do not provide much value unless they are grouped together based on a semantic view. The challenge of course is that since we monitor changes at the block layer, file-system level objects are not visible to us. Hence, we must recreate the relationships between blocks in lieu of file-level information. Fortunately, all hope is not lost as file-systems use various mechanisms to describe data layout on disk. This layout includes how files, directories and other system objects are mapped to blocks on disk. In addition, these structures are kept at set locations on disk and have a predefined binary format. As our main deployment scenario is the enterprise model, like Payne *et al.* [31] we assume that the file-system (e.g., ext3, ntfs, etc.) in use by the guest VM is known.

Armed with that knowledge, the storage module periodically scans the disk to find the inodes and superblocks[6] so that this meta-data can be used during forensic recovery. That is, for any set of blocks returned by a `report`() operator, we use the stored file-system metadata to map a cluster of blocks to files. For ease of use, we also provide a facility that allows an analyst to provide a list of hashes of files and their corresponding filenames. The `report`() operators use that information (if available) to compare the hashes in the list to those of the recreated files, and tags them with the appropriate filename.

5. EMPIRICAL EVALUATION

While having the ability to record fine-grained data accesses is a useful feature, any such system would be impractical if the approach induced high overhead. In what follows, we provide an analysis of our accuracy and overhead. Our experiments were conducted on an Intel Core2 Dual Core machine running at 2.53GHz with Intel-VT hardware virtualization support enabled. The total memory installed was 2GB. Xen 3.4 with HVM support and our modifications served as the hypervisor, and the guest virtual machine was either Windows XP (SP2) or Debian Linux (kernel 2.6.26). The virtual machine was allocated 512 MB of memory and had two disks mounted, a 20GB system disk and 80GB data disk. The 80GB data disk hosted the user's home directories, and was mounted as a monitored virtual disk. Therefore, all blocks in this virtual disk were automatically added to the watchlist of the storage module. The virtual machine was allocated 1 virtual CPU, and in all experiments the hyper-

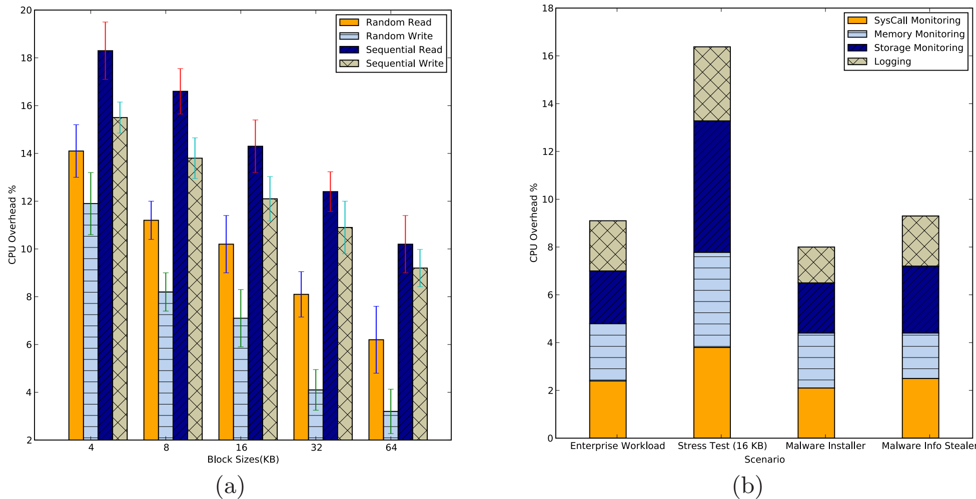

Figure 5: Runtime overhead for (a) varying block sizes and access patterns, and (b) across different test scenarios.

visor and the virtual machine were pinned to two different physical cores. We do so in order to reflect accurate measurements of CPU overheads.

First, the overhead associated with our approach was calculated under a stress test using a `Workload Generator` and a workload modeled for `Enterprise` users. Specifically, we subjected our design to a series of tests (using IOMeter)[7] to study resource utilization under heavy usage, and used a scripting framework for Windows (called AutoIt) to automate concurrent use of a variety of applications. The application set we chose was Microsoft Office, plus several tools to create, delete, and modify files created by the Office applications. The parameters for the workload generator (e.g., the number of concurrent applications, average typing speed, frequency of micro-operations including spell-check in Word and cell calculations in Excel, etc.) were set based on empirical studies [35, 17]. The `Workload Generator` tests were conducted on an empty NTFS partition on the data disk, while the `Enterprise Workload` was tested with pre-seeded data comprising a set of Microsoft Office files along with additional binaries. These binaries performed various memory mapped, network and shared memory operations. The binaries were added to increase the pool of applications loaded during the tests, and hence add greater dynamism in the resulting code pages loaded into memory.

Runtime Overhead. Our runtime overhead is shown in Figure 5(a). The block sizes were chosen to reflect normal I/O request patterns, and for each block size, we performed random read, random write, sequential read and sequential write access patterns. The reported result is the average and variance of 10 runs. Each run was performed under a fresh boot of the guest VM to eliminate any disk cache effects. The IOMeter experiments were run on the same data disk with and without the monitoring code, and the overhead was calculated as the percent change in CPU utilization. The CPU utilization was monitored on both cores using performance counters. The reported utilization is the normalized sum of both cores.

Not surprisingly, writes have a lower overhead due to the increased time for completion from the underlying disk. Conversely, sequential access consumes more CPU as the disk subsystem responds faster in this case, and hence the I/O ring is quickly emptied by the hypervisor. Even under this *stress* test, the overhead is approximately 18%. This moderate overhead can be attributed to several factors in our design, including the scheduling of lazy writes of our data structures, the lightweight nature of our system-call monitoring, and the efficiency of the algorithms we use to extract the code pages.

Figure 5(b) shows a more detailed breakdown of CPU overhead as consumed by the different modules. Notice that the majority of the overhead for the stress test (for the 16KB case) can be attributed to the storage subsystem, as many of the accesses induced in this workload are for blocks that are only accessed once. We remind the reader that the expected use case for our platform is under the `Enterprise-Workload` model and the overall overhead in this case is below 10%, with no single module incurring overhead above 3%. Also shown are the averaged overheads induced when monitoring and logging the activities of several real-world malware. In all cases, the overload is below 10%, which is arguably efficient-enough for real-world deployment. We return to a more detailed discussion of how we reconstructed the behavioral profiles of these malware using our forensic platform in Section 6.

Another important dimension to consider is the growth of the log compared to the amount of actual data written by the guest VM. Recall that the audit log stores an initial copy of a block at the first time of access, and thenceforth only stores the changes to that block. Furthermore, at every snapshot, merging is performed and the data is stored on disk in an optimized binary format.

We examined the log file growth by monitoring the audit log size at every purge of the version-trees to disk (10 mins in our current implementation). In the case of the `Enterprise Workload` , the experiment lasted for 1 hour, with a minimum of 4 applications running at any point in time. During the experiment, control scripts cause the overall volume of files to increase at a rate of at least 10%. The

Malware	% Activity in Log	Disk search	Exfiltration	Classification
Zeus & Variants	35.0	active	active	info stealer
Ldpinch	22.5	active	active	info stealer
Alureon	15.0	active	active	info stealer
Koobface	10.0	passive	active	installer
Bubnix	5.0	passive	active	installer
Sinowal	4.0	active	active	both
Conpro	3.5	active	active	installer
Vundo	3.0	passive	passive	installer
Rustock	1.5	passive	passive	installer
Slenfbot	0.5	passive	passive	installer

Table 1: Malicious applications recovered from the audit log, and their high-level classification.

file sizes of the new files were chosen from a zipf distribution, allowing for a mix of small and large files [25]. We also included operations such as `make` to emulate creation and deletion of files. The overhead (i.e. additional disk space used to store logs and metadata compared to the monitored disk blocks) was on average $\approx 2\%$. Since the `Enterprise-Workload` is meant to reflect day-to-day usage patter this low overhead indicated that this platform is practical and deployable.

Accuracy of Reconstruction. To examine the accuracy of our logging infrastructure, we explore our ability to detect accesses to the monitored data store by "unauthorized" applications. Again, the `Enterprise Workload` was used for these experiments, but with a varying concurrency parameter. Specifically, each run now included a set of authorized applications and a varying percentage of other applications that also performed I/O operations on monitored blocks. The ratio of unauthorized applications for a given run was increased in steps of 5%, until all applications running were unauthorized. The task at hand was to reconstruct all illicit accesses to the disk. The illicit accesses include copying a file into memory, sending a file over a network connection, and shared memory or IPC operations on monitored objects. The audit log was then queried for the time-window spanning the entire duration of the experiment to identify both the unauthorized applications and the illicit access to blocks. The system achieved a true positive rate of 95% for identification of the illicit applications and a 96% true positive rate in identifying the blocks accessed by these applications.

6. REAL-WORLD CASE STUDY

To further showcase the benefits of our platform, we report on our experience with deploying our framework in an open-access environment that arguably reflects the common case of corporate laptops being used in public WiFi environments. Specifically, we deployed our approach on a laptop supporting hardware virtualization, on top of which we ran a Windows XP guest with unfettered access to the network. The enterprise workload was configured to run on the guest system to simulate a corporate user. The monitored area was set to be the entire virtual disk exposed in the guest (roughly 4.0 GBs of storage). While there was no host or network-level intrusion prevention system in place on the guest system, we also deployed Snort and captured network traffic on a separate machine. This allowed us to later confirm findings derived from our audit mechanism. The laptop was left connected to the network for one week, and its outbound traffic was rate-limited in an attempt to limit the use of the machine to infect other network citizens.

To automate the forensic recovery process, we make use of a proof-of-concept tool that mines the audit logs looking for suspicious activity. Similar to Patagonix [27] we assume the existence of a trusted external database, $\mathcal{D}$, (e.g., [30]) that contains cryptographic hashes of applications the system administrator trusts. The code pages for these authorized applications were created using a userland application that runs inside a pristine VM and executes an automated script to launch applications. The userland application communicates with the memory monitoring module, and tags the pages collected for the current application. The pages are extracted as described in Section 3.1.3, and are stored along with the application tags. Notice that these mappings only need be created once by the system administrator.

We then mined the log for each day using `report`(24hr, $\mathcal{B}$) to build a set of identifiers ($p \in P$), where $\mathcal{B}$ ={blocks for the `temp`, `system`, `system32` directories and the master boot record}. Next, we extracted all causally related activity for each $p \notin \mathcal{D}$, by issuing `report`(24hr, causal, p). The result is the stored blocks that relate to this activity. These blocks are automatically reassembled by mapping blocks to files using the filesystem metadata saved by the storage module (as discussed in Section 4.1). At this point we have a set of unsanctioned applications and what blocks they touched on disk. For each returned event sequence, we then classified it as either (i) an *info stealer*: that is, a process that copied monitored objects onto an external location (e.g., L=network) or (ii) an *installer*: a process that installs blocks belonging to an info stealer.

To do so, our recovery utility first iterates through the set of unsanctioned applications and checks the corresponding version-trees for events that match an info stealer's signature. For each match, we extract all its blocks, and issue `report`(24hr, $b_i, \ldots, b_n$). This yields the list of all unsanctioned applications that touched an info stealer's blocks. From this list, we searched for the one that initially wrote the blocks onto disk by issuing `report`(24hr, write, $b_i, \ldots, b_n$). The result is an installer.

Table 1 shows the result of running our proof-of-concept forensic tool on the audit logs collected from the laptop. The table shows the percentage of activity for each malicious binary and the classification as per the tool. For independent analysis, we uploaded the reconstructed files to Microsoft's Malware Center; indeed all the samples were returned as positive confirmation as malware. We also subjected the

entire disk to a suite of AV software, and no binaries were flagged beyond those that we already detected by our tool.

To get a better sense of what a recovered binary did, we classify its behavior as active if it had activity in the audit logs every day after it was first installed; or passive otherwise. The label "Exfiltration" means that data was shipped off the disk. "Disk search" means that the malware scanned for files on the monitored store. As the table shows, approximately 70% of the recorded activity can be attributed to the info stealers. Upon closer examination of the blocks that were accessed by these binaries, we were able to classify the files as Internet Explorer password caches and Microsoft Protected Storage files. An interesting case worth pointing out here is Zeus. The causal event linkage by the forensic tool allowed us to track the initialization of Zeus as `Zbot` by `Sinowal`. Even though `Sinowal` constitutes only 4% of activity in the logs, it was responsible for downloading 60% of the malware on the system. Zeus appears to be a variant that used Amazon's EC-2 machines as control centers[8].

Interestingly, the average growth of our audit log was only 15 MB per day compared to over 200 MB per day from the combined Snort and network data recorded during the experiment. Yet, as we show later, the data we are able to capture is detailed enough to allow one to perform interesting behavioral analyses. The analysis in Table 1 took less than 4 hours in total to generate the report, and our proof-of-concept prototype can be significantly optimized.

6.1 Example Reconstruction

With the framework at our disposal, we decided to explore its flexibility in helping with behavioral analysis. Specifically, we were interested in analyzing `Mebroot`, which is a part of the stealthy `Sinowal` family. `Mebroot` serves as a good example as reports by F-Secure [8] labels it as one of the "stealthiest" malware they have encountered because it eschews traditional windows system call hooking, thereby making its execution very hard to detect. The anatomy of the `Mebroot` attack can be summarized as follows: first, a binary is downloaded and executed. Next, the payload (i.e., from the binary) is installed, and the master boot record (MBR) is modified. Lastly, the installer deletes itself.

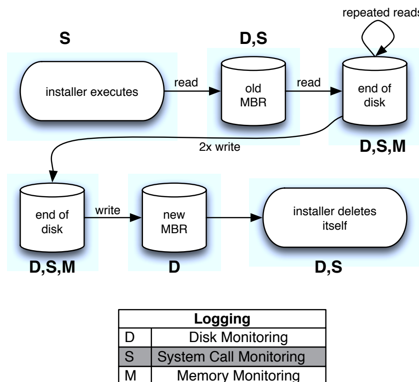

Figure 6: Annotated graph of the causal reconstruction of `Mebroot`'s attack vector as recovered from processing the audit logs

To understand what `Mebroot` did, we issued `report`(∞, causal, ID(`Mebroot`)). The reason why the causal relationship between the first two steps is built by our monitoring infrastructure should be obvious. In our platform, the connection between the first and last steps is made when the file deletion is noted (i.e., when the storage module rescans the inodes). An annotated profile of the behavior recovered from our audit log is shown in Figure 6. Notice that because we store "diffs" in the version trees, we are also able to see all the modifications made to the MBR.

To further evaluate the strength of our platform in helping an analyst quickly reconstruct what happened after a compromise is detected, we provided two malware samples to a seasoned malware analyst (i.e., the second author) for inspection. In both cases, the malware was successfully unpacked and disassembled using commercial software and inspected using dynamic analysis techniques for system-call sequence analysis, for finding the payload in memory, and for single-stepping its execution. We then compared our results to those from this labor-intensive exercise.

Syscall %	Phalanx2		`Mebroot`	
	Manual	Forensic	Manual	Forensic
Storage	0.72	0.68	0.91	0.95
Memory	0.26	0.30	0.08	0.05
Other	0.02	0.02	0.01	0.0

Table 2: Comparison of the profiles created by manual analysis vs. reconstruction using our platform

The breakdown in terms of diagnosed functionality is shown in Table 2. The overall results were strikingly similar, though the analyst was able to discover several hooks coded in Phalanx2 (a sophisticated info stealer) for hiding itself, the presence of a backdoor, and different modes for injection that are not observable by our platform. From a functional point of view, the results for `Mebroot` were equivalent. More important, however, is the fact that the manual inspection verified the behavioral profile that we reported, attesting to the accuracy of the linkages we inferred *automatically*.

7. ATTACKS AND LIMITATIONS

As stated earlier, the approach we take relies on the security properties of the hypervisor to properly isolate our monitoring code from tampering by malicious entities residing in the guest OS's. This assumption is not unique to our solution, and to date, there has been no concrete demonstration that suggests otherwise. However, if the security of the hypervisor is undermined, so too is the integrity and correctness of the transactions we record. Likewise, our approach suffers from the same limitations that all other approaches that have extended Xen (e.g., [31, 7, 22, 18]) suffer from— namely, that it extends the trusted code base.

A known weakness of current hypervisor designs is their vulnerability to hypervisor-detection attacks [11, 5, 13]. One way to address these attacks might be to rely on a thin hypervisor layer built specifically for data forensics, instead of using a hypervisor like Xen which provides such a rich set of functionality (which inevitably lends itself to being easily detected). Once the presence of a hypervisor has been detected, the attacker can, for instance, change the guest VM's state in a way that would cause the forensic platform to capture a morphed view of the VM [13]. An example of such an attack would involve the attacker attempting to circumvent our event model by modifying the System Call Tables in Linux or the SSDT in Windows to remap system

calls. This could cause the framework to trigger false events at the system call layer and pollute the audit logs. That said, such an attack poses a challenge for all the hypervisor-based monitoring platforms we are aware of. Techniques to mitigate such attacks remain an open problem.

Resource exhaustion attacks offer another avenue for hindering our ability to track causal chains. As our infrastructure tracks all monitored objects in memory, an attacker could attempt to access hundreds of files within a short period of time, causing the memory monitoring module to allocate space for each object in its watchlist. If done using multiple processes, the attack would likely lead to memory exhaustion, in which case some monitored objects would need to be evicted from the watchlist. While we have built several optimizations to mitigate such threats (e.g., by collapsing contiguous pages to be tracked as a single address range), this attack strategy remains viable.

Lastly, since we do not monitor interactions that *directly* manipulate the receive and transmit rings of virtual network interfaces (NICs), such accesses will not be logged. Moreover, our current prototype only logs accesses to monitored blocks, and does not prevent such accesses. That said, extending our approach to cover these rings and/or to block unauthorized accesses is largely an engineering exercise that we leave as future work.

8. CONCLUSION

We present an architecture for efficiently and transparently recording the accesses to monitored objects. Our techniques take advantage of characteristics of platforms supporting hardware virtualization, and show how lightweight mechanisms can be built to monitor the causal data flow of objects in a virtual machine—using only the abstractions exposed by the hypervisor. The heuristics we developed allow the monitoring framework to coalesce the events collected at various layers of abstraction, and to map these events back to the offending processes. The mappings we infer are recorded in an audit trail, and we provide several mechanisms that help with data forensics efforts; for example, allowing an analyst to quickly reconstruct detailed information about what happened when such information is needed the most (e.g., after a system compromise). To demonstrate the practical utility of our framework, we show how our approach can be used to glean insightful information on behavioral profiles of malware activity after a security breach has been detected.

9. CODE AVAILABILITY

The source code for both the monitoring platform (i.e., patches to Xen) and our packaged tools are available on request under a BSD license for research and non-commercial purposes. Please contact the first author for more information on obtaining the software.

Acknowledgments

We thank the anonymous reviewers for their insightful comments and suggestions for improving earlier drafts of this paper. This work is supported in part by the National Science Foundation under awards CNS-0915364 and CNS-0852649.

10. REFERENCES

[1] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the Art of Virtualization. In *Proceedings of the* 19th *ACM Symposium on Operating Systems Principles* (2003), pp. 164–177.

[2] Buchholz, F., and Spafford, E. On the Role of File System Metadata in Digital Forensics. *Digital Investigation 1*, 4 (2004), 298 – 309.

[3] Chen, P., and Noble, B. When Virtual is Better than Real. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (May. 2001), pp. 133–138.

[4] Chen, S., Xu, J., Nakka, N., Kalbarczyk, Z., and Iyer, R. K. Defeating Memory Corruption Attacks via Pointer Taintedness Detection. In *IEEE International Conference on Dependable Systems and Networks (DSN* (2005), pp. 378–387.

[5] Chen, X., Andersen, J., Mao, Z., Bailey, M., and Nazario, J. Towards an Understanding of Anti-virtualization and Anti-debugging Behavior in Modern Malware. In *Dependable Systems and Networks* (June 2008), pp. 177–186.

[6] Denning, D. E., and Denning, P. J. Certification of Programs for Secure Information Flow. *Communications of the ACM 20*, 7 (1977), 504–513.

[7] Dinaburg, A., Royal, P., Sharif, M., and Lee, W. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (2008), pp. 51–62.

[8] F-Secure. MBR Rootkit, A New Breed of Malware. *See* `http://www.f-secure.com/weblog/archives/00001393.html` (2008).

[9] Farmer, D., and Venema, W. *Forensic Discovery*. Addison-Wesley, 2006.

[10] Franklin, J., Perrig, A., Paxson, V., and Savage, S. An Inquiry into the Nature and Causes of the Wealth of Internet Miscreants. In *Proceedings of the* 14th *ACM conference on Computer and communications security* (2007), pp. 375–388.

[11] Garfinkel, T., Adams, K., Warfield, A., and Franklin, J. Compatibility is not Transparency: VMM Detection Myths and Realities. In *Proceedings of the* 11th *USENIX workshop on Hot topics in operating systems* (2007), pp. 1–6.

[12] Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., and Boneh, D. Terra: A Virtual Machine-Based Platform for Trusted Computing. In *Proceedings of ACM Symposium on Operating System Principles* (2003), pp. 193–206.

[13] Garfinkel, T., and Rosenblum, M. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Network and Distributed Systems Security Symposium* (2003), pp. 191–206.

[14] Goel, A., Po, K., Farhadi, K., Li, Z., and de Lara, E. The Taser Intrusion Detection System. In *Proceedings of Symposium on Operating Systems Principles* (Oct. 2005).

[15] Goldberg, R. Survey of Virtual Machine Research. *IEEE Computer Magazine 7*, 6 (1974), 34–35.

[16] JAIN, S., SHAFIQUE, F., DJERIC, V., AND GOEL, A. Application-Level Isolation and Recovery with Solitude. In *Proceedings of EuroSys* (Apr. 2008), pp. 95–107.

[17] JAY, C., GLENCROSS, M., AND HUBBOLD, R. Modeling the Effects of Delayed Haptic and Visual Feedback in a Collaborative Virtual Environment. *ACM Transactions on Computer-Human Interaction 14*, 2 (2007), 8.

[18] JIANG, X., WANG, X., AND XU, D. Stealthy Malware Detection through VMM-based "out-of-the-box" Semantic View Reconstruction. In *Proceedings of the* 14th *ACM conference on Computer and Communications Security* (2007), pp. 128–138.

[19] JONES, S. T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Antfarm: Tracking Processes in a Virtual Machine Environment. In *Proceedings of the USENIX Annual Technical Conference* (2006).

[20] JONES, S. T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Geiger: Monitoring the Buffer Cache in a Virtual Machine Environment. *SIGPLAN Not. 41*, 11 (2006), 14–24.

[21] KIM, G. H., AND SPAFFORD, E. H. The Design and Implementation of Tripwire: a File System Integrity Checker. In *Proceedings of the* 2nd *ACM Conference on Computer and Communications Security* (1994), ACM, pp. 18–29.

[22] KING, S., AND CHEN, P. Backtracking Intrusions. *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles* (Dec 2003).

[23] KING, S. T., MAO, Z. M., LUCCHETTI, D. G., AND CHEN, P. M. Enriching intrusion alerts through multi-host causality. In *Proceedings of Network and Distributed System Security Symposium* (2005).

[24] KRISHNAN, S., AND MONROSE, F. Time Capsule: Secure Recording of Accesses to a Protected Datastore. In *Proceedings of the* 2nd *ACM Workshop on Virtual Machine Security* (Nov. 2009).

[25] LEUNG, A. W., PASUPATHY, S., GOODSON, G., AND MILLER, E. L. Measurement and Analysis of Large-scale Network File System Workloads. In *USENIX Annual Technical Conference* (2008), pp. 213–226.

[26] LEUNG, F., NEIGER, G., RODGERS, D., SANTONI, A., AND UHLIG, R. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. *Intel Technology Journal 10* (2006).

[27] LITTY, L., LAGAR-CAVILLA, H., AND LIE, D. Hypervisor Support for Identifying Covertly Executing Binaries. In *Proceedings of USENIX Security Symposium* (Aug. 2008), pp. 243–257.

[28] MUNISWAMY-REDDY, K., HOLLAND, D., BRAUN, U., AND SELTZER, M. Provenance-aware Storage Systems. In *Proceedings of the 2006 USENIX Annual Technical Conference* (2006), pp. 43–56.

[29] MUNISWAMY-REDDY, K.-K., MACKO, P., AND SELTZER, M. Provenance for the Cloud. In *USENIX Conference on File and Storage Technologies (FAST)* (Berkeley, CA, USA, 2010), USENIX Association.

[30] NIST. National Software Reference Library, 2009.

[31] PAYNE, B. D., CARBONE, M., AND LEE, W. Secure and flexible monitoring of virtual machines. *Annual Computer Security Applications Conference* (2007), 385–397.

[32] PROVOS, N., MCNAMEE, D., MAVROMMATIS, P., WANG, K., AND MODADUGU, N. The Ghost in the Browser: Analysis of Web-based Malware. In *First Workshop on Hot Topics in Understanding Botnets* (2006).

[33] QUINLAN, S., AND DORWARD, S. Venti: A New Approach to Archival Data Storage. In *Proceedings of the USENIX Conference on File and Storage Technologies* (2002), pp. 89–101.

[34] SEAN PEISET AND MATT BISHOP AND KEITH MARZULLO. Computer Forensics in Forensis. *ACM Operating System Review 42* (2008).

[35] SHNEIDERMAN, B. Response Time and Display Rate in Human Performance with Computers. *ACM Computing Surveys 16*, 3 (1984), 265–285.

[36] SLOWINSKA, A., AND BOS, H. Pointless Tainting? Evaluating the Practicality of Pointer Tainting. In *Proceedings of EuroSys* (Apr. 2009).

[37] VINCENZETTI, D., AND COTROZZI, M. ATP - Anti Tampering Program. In *Proceedings of USENIX Security* (1993), pp. 79–90.

Notes

[1]The `SYSENTER` call on the Intel platform uses the `MSR SYSENTER_EIP` to find the target instruction. This `MSR` is always located on Intel machines at address 176h.

[2]System call numbers are pushed into `eax`

[3]Recall the memory module must translate the guest virtual address to its physical address in a machine physical page

[4]The destination machine physical page in `memcpy`

[5]Specifically, a hidden page is appended in the shadow page table of the process with the timestamp and objects accessed

[6]Similarly, the Master File Table and Master File Records under NTFS.

[7]See `http://www.iometer.org`

[8]We verified this hypothesis independently based on our network logs.