SAV-V: Securing Anti-Virus with Virtualization*

Jacob R. Lorch
Microsoft Research

Bryan Parno
Carnegie Mellon University

Helen J. Wang
Microsoft Research

Abstract

Today's desktop PCs rely on security software such as anti-virus products and personal firewalls for protection. Unfortunately, malware authors have adapted by specifically targeting and disabling these defenses, a practice exacerbated by the rise in zero-day exploits. In this paper, we present the design, implementation, and evaluation of SAV-V, a platform that enhances the detection capabilities of anti-virus software. Our platform leverages virtualization to preserve the *integrity* of AV software and to guarantee access to AV updates. SAV-V also uses secure logging and a split file system to preserve the *fidelity* of input to the AV program. Combined with our technique of fake shutdowns, these measures allow SAV-V to eventually detect any zero-day malware that writes to disk. Benchmarks of our prototype system suggest that SAV-V can be implemented efficiently, and we validate our prototype by testing it against real-world malware.

1 Introduction

The typical desktop PC depends on various forms of protection software, including anti-virus (AV) and anti-spyware applications, as well as host-based firewalls. The anti-malware tools safeguard persistent state on the PC, while firewalls cleanse network input. To provide protection, these tools rely on rules and signatures developed based on knowledge of malware, attacks, and software vulnerabilities. While these techniques do not offer perfect protection, they represent the primary defense mechanism for millions of computer users.

Unfortunately, even with these protection mechanisms in place, most computers remain vulnerable to zero-day attacks based on undiscovered vulnerabilities or unknown malware, and indeed zero-day attacks are likely to be a fact of life for years to come. Recent trends indicate that zero-day exploits are on the rise [33]. As new technologies [28, 38] are deployed to defend against known vulnerabilities, the incentive to launch zero-day exploits will increase. As a result, future computer systems must be able to deal with, or at least recover from, zero-day attacks.

Zero-day attacks fundamentally undermine a user's confidence in the security of her machine, since they can seize control of applications and even the operating system and then use this control to disable or subvert protection software. This subversion can be subtle and thus difficult to detect. For example, it may leave the protection software running but prevent it from downloading updates needed to detect and remove the infection. By keeping a low profile, malware may remain undetected indefinitely, and throughout this time the user is unwittingly vulnerable to arbitrary malicious activity. For instance, her bank passwords may be captured, or her computer may be used to send spam or launch denial-of-service attacks.

In this work, we aim to create a platform, SAV-V, that preserves the detection capabilities of protection software, even in the presence of zero-day attacks that can subvert the operating system. Note that we do not attempt to stop all zero-day attacks from executing; developing a practical system for reliably preventing the execution of unknown malware targeting unknown exploits remains an open research problem. Instead,

*Based on research conducted at Microsoft Research from June 2006 through April 2007.

SAV-V enables AV software to detect past zero-day infections, even ones that attempted to disrupt the AV software itself, when signatures for those infections become available. Compared with today's AV techniques, which leave the user perpetually uncertain as to the trustworthiness of her computer, the ability to reliably detect intrusions past or present represents a significant step forward. Once the user is informed of an intrusion, she can terminate her use of the corrupted system and take steps to recover the system to a known-good state. These steps can involve rollback to a previous backup or more advanced techniques that allow selective replay of legitimate activity [16, 20].

By focusing on protection software, we adopt a hierarchical protection model in which the SAV-V platform secures the protection software and the protection software secures the rest of the system. This approach allows AV vendors to continue in their traditional role as the source of creativity in signature development and deployment. As a result, the SAV-V model is backward-compatible with today's AV operations and allows easy deployment.

Our hierarchical protection model naturally suggests leveraging the extra privilege level provided by Virtual Machine Monitors (VMM). We draw two lessons from our exploration in the solution space of a VM-based SAV-V platform. First, *VM introspection-based approaches have significant security risks*: VM introspection infers high-level software semantics (such as file accesses) from the hardware-level state exposed by the monitored VM. To do so, assumptions must be made about the OS and software structures running in the monitored VM. However, these assumptions can be violated by malware intrusions. The second lesson is that *securing an application's integrity through OS hardening requires excessive complexity and cannot realistically be complete*, since this entails securing the enormous number of OS components on which the application depends.

Our design for SAV-V is based on these lessons. To make the AV software tamperproof without the need for OS hardening, we isolate AV software in a secure VM separate from the guest VM that the AV software protects. To eliminate the security risks of VM introspection, we have the AV software inspect the guest's state at the file system level rather than the hardware level.

Driven by two attacker models, we have designed, implemented, and evaluated on Windows two file-system inspection approaches for the SAV-V platform: Guest-Initiated Logging (GIL) and Split File System (Split FS). The former targets file-based attacks in which the system is compromised as the result of executing a malicious file. This attack vector encompasses most of today's viruses and does not necessarily depend on the existence of software vulnerabilities, since social engineering techniques can convince users to execute malicious binaries. As Pennington et al. note [25], the vast majority of today's malware, particularly on home PCs, can be detected at the disk level. Split FS offers a higher level of assurance, guaranteeing that the AV software can, if desired, monitor and/or interpose on every file-system operation, even if the guest operating system is subverted by a memory-based attack.

While Split FS offers stronger guarantees, it incurs substantially more overhead than GIL. GIL only crosses the virtual machine boundary when the file system is modified, and only adds a few microseconds to file-system operations. In contrast, Split FS must cross the virtual machine boundary even on reads, and can add more than a millisecond to file-system operations. When we compare these two implementations with a file-system-intensive macrobenchmark, we find that GIL increases completion time by 5% while Split FS increases it by 62%. These results illustrate the inherent tradeoff between security and performance.

With Split FS in place, attackers who exploit memory-based vulnerabilities will likely postpone writing to disk for as long as possible. However, they must eventually write to disk to persist across a shutdown. To speed detection of such malware that postpones suspicious writes until shutdown, we propose the use of random *fake shutdowns*. We present our design for this in Section 5.4.

The paper is structured as follows. In Section 2, we present our goals for SAV-V. Then, in Section 3 we present a taxonomy of malware, which we use to clearly define the types of malware SAV-V can and cannot defend against. Section 4 explains our design rationale, and Section 5 describes the architecture for the SAV-V platform. We present our security analysis in Section 6. We then describe the details of our

implementation and evaluation in Section 7 and suggest directions for future research in Section 8. Finally, we survey related work in this area in Section 9 and offer our conclusions in Section 10.

2 Goals of SAV-V

AV software detects and removes malware by comparing newly created or modified files against virus signatures. AV detection and cleansing takes place both at boot time and during normal operations. For the latter, AV software intercepts file-system API calls so that it can scan files when they are first written to disk or first accessed.

The SAV-V platform should endow AV software with the following capabilities:

- *Prevent New Infections by Known Viruses.* AV software should be able to prevent new infections that can be detected by the current set of virus signatures. To provide this property, the AV software must see the authentic file system at all times, and it must execute with integrity, i.e., it cannot be tampered with or disabled.
- *Detect Past Zero-Day Infections.* AV software should be able to detect past zero-day infections when signatures for the associated malware become available. This requires the AV software to continue functioning correctly after the zero-day infection.
- *Obtain Updated Signatures.* Malware should not be able to tamper with the network connection that AV software uses to receive updates. Note that we consider only malware resident on the computer itself; defending against an adversary who floods the victim's network link from an external computer is beyond the scope of this paper.

Zero-day attacks prevent current AV programs from fully achieving any of these capabilities. Since AV programs typically cannot stop the execution of zero-day attacks, the attacker can disable the AV software and prevent it from detecting known malware that later infects the system. The malware may instead allow the AV software to continue executing, but prevent it from receiving the updates needed to detect the new attack. In either case, the AV software offers little utility to the user, and as a result, she will have correspondingly little faith in the protection it provides.

3 Malware Taxonomy and SAV-V's Scope

To describe the types of malware our techniques can defend against, we create a taxonomy of malware based on its interaction with the disk. The classes and subclasses are described in detail below.

3.1 Disk-based Malware

Disk-based malware must be written to disk before it can execute. The class of disk-based malware includes typical viruses that spread via e-mail or infected files, often relying on social engineering tricks to persuade users to execute them. As a result, even operating systems without any vulnerabilities may still allow disk-based malware to execute. The MyDoom virus is a prototypical example [8].

AV software is most effective against disk-based malware. As long as it has an appropriate signature for the malware, and it has not yet been compromised by zero-day malware, it can usually prevent files containing that malware from executing.

3.2 Memory-based Malware

Memory-based malware can execute without being written to disk. Typically, it exploits a software vulnerability such as a buffer overflow, format-string vulnerability, etc. Malware in this class rarely requires

active participation from the user. Examples in this class include the Blaster and SQL Slammer worms.

We classify memory-based malware into one of two subclasses: *pure* memory-based malware, which never writes anything to disk, and *mixed* memory-based malware, which does write to disk. Below, we describe the class of mixed memory-based malware in more detail.

Memory-based malware authors have various reasons for writing to disk. First, *convenience*—rather than write new tools that avoid disk writes, they may use pre-existing tools, such as an ftp client, that write to disk. Second, *size constraints*—they may need to download large amounts of data that will not fit in main memory. For example, if the malware is to serve as a repository for bootleg movies or pirated software, it may need to store them on disk. Third, and most importantly, *persistence*—malware authors often want their malware to maintain control of the system beyond a shutdown. The fact that pure memory-based malware is purged from the system upon shutdown provides a strong motivation to write to disk.

An example of mixed memory-based malware is the Blaster worm, which exploited a DCOM RPC vulnerability in Microsoft Windows to gain execution privileges, then downloaded an executable to the infected computer [7].

AV software generally does not protect against memory-based exploits, but it may be able to detect mixed memory-based malware. Note, however, that mixed memory-based malware with the proper privileges and capabilities may disrupt current AV software, then write to disk without risk of detection.

3.3 Scope of SAV-V

We have designed SAV-V to defend against both disk-based malware and mixed memory-based malware. At present, we leave consideration of pure memory-based malware for future work. Prior work suggests that virtually all modern malware leaves traces of its presence on disk [25], indicating that SAV-V can be effective even without addressing pure memory-based malware.

4 Design Rationale

In this section, we discuss approaches for our platform design that we considered and rejected before settling on the SAV-V architecture. The flaws we found served as useful guidance in the design of SAV-V.

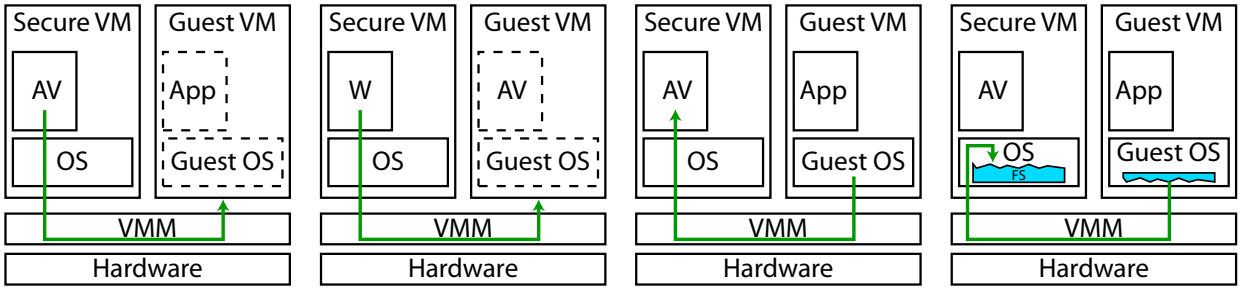
4.1 Virtualization-Based Approaches

Virtualization technology permits strong isolation of software components, and thus is a logical component for our system of hierarchical protection.

A virtual machine monitor (VMM) is a thin layer of software that typically runs directly on the physical machine while presenting the abstraction of multiple virtual machines (VMs). Within each virtual machine, an operating system runs as it normally would on a physical machine, often unaware of the virtualization layer underneath. Popular examples of virtualization software for commodity desktops include Xen [6], various products from VMWare [36], and Microsoft's Virtual PC [22].

An important property provided by virtualization is *isolation*: software inside one VM cannot see or affect another VM unless explicitly permitted by the VMM. This suggests a natural way to achieve our hierarchy of protection. We can place most of the system to be protected into one VM, called the *guest VM*, and the protection software itself into another VM, called the *secure VM*. The VMM permits the secure VM to monitor and control the guest VM, but does not permit the reverse. Thus, malware that takes control of the guest VM cannot disrupt the protection software.

VM introspection is a powerful technique that enables one VM to monitor another [5,15,19]. Essentially, the VMM allows one VM to examine the hardware state of another VM. The inspecting VM can then use this hardware state to infer the high-level software state of the inspected VM. Often, the VMM also allows the



(a) **Disk Introspection** (b) **Watcher** (c) **Guest-Initiated Logging** (d) **Split File System**

Figure 1: *Figures 1(a) and 1(b) illustrate two approaches we considered and rejected. Figures 1(c) and 1(d) show the two inspection techniques used by SAV-V. AV indicates the anti-virus application, and W signifies a watcher program.*

inspecting VM to register a callback that gets invoked when certain hardware events occur on the inspected VM. This way, the inspecting VM can continually monitor important elements of the inspected VM's state.

4.1.1 Disk Introspection

The first approach we considered places the user's primary OS and applications in a guest VM and the AV software in a secure VM from which it can use VM introspection to perform its tasks. Figure 1(a) illustrates this approach.

With disk introspection, the VMM provides the necessary hooks and permissions to allow the secure VM to inspect the disk state of the guest VM, and to be notified each time the guest OS writes to disk. As noted in Section 9, prior work employs a similar approach for providing host-based intrusion detection [15].

The primary problem with disk introspection is the large number of assumptions it must make about how the software inside the guest VM behaves. The VMM can only provide accurate information about the low-level hardware machine state of the guest VM. Any additional information must be inferred. So, the VMM can provide information about blocks written to disk, but the security software must somehow use that information to infer information about file writes. While some researchers have considered the problem of inferring file system information from block-level events [30], their work focuses on performance optimizations and does not consider the possibility of an adversarial environment. Indeed, once malware in the guest VM succeeds in executing, it can immediately violate the assumptions used by the security software when it makes its inferences. Since a compromise of the guest OS will make the information gleaned from VM introspection unreliable, we cannot rely on this approach to safeguard security software.

4.1.2 Watcher

Another approach we considered we called the *watcher* approach. In this approach, the security software runs inside the guest VM, not the secure VM. What runs in the secure VM is a watcher program, which uses VM introspection to ensure the integrity of the security software's execution and the fidelity of the information it receives (see Figure 1(b)). In the end, this approach proved excessively complex and unwieldy.

Placing the security software inside the guest VM gives it direct visibility into the guest software state. It no longer needs to make inferences about the correspondence of this state to the guest hardware state. It also eliminates the need to modify or shim the security software, since it continues its normal operations.

The watcher program resides in the secure VM and monitors the execution of the security software. It verifies that the security software has not been tampered with, and it also guarantees that the security software observes the correct information. In theory, since the watcher program only monitors a single

piece of software, it should require a minimal amount of VM introspection, while the security software's position in the guest VM gives it an accurate view of the relevant guest operations.

The problem with this approach is that since the security software operates inside the guest VM, it depends heavily on the guest OS. As a result, safeguarding the operation of the security software requires safeguarding the transitive closure of every piece of the operating system that the security software relies on, including both code and data. For example, to ensure that the security software is run with sufficient frequency, we must secure the scheduler. The scheduler, in turn, relies on the timing devices, various interrupt handlers, and the state stored in each thread that indicates its priority and the amount of time it has already run. Similarly, an AV program relies on the file system code, which in turn relies on the integrity of various caches, e.g., the block cache, the directory entry cache, etc. Thus, the watcher's task of "simply" protecting the security software actually involves the enormous subproblem of protecting the code and data of most of the operating system. Given the size and complexity of modern general-purpose operating systems, this is unlikely to be the optimal method for protecting security software.

4.2 A Non-Virtualization-Based Approach

As an alternative to leveraging the higher privilege level of a VMM, we considered an approach using the higher privilege level enjoyed by the BIOS. This alternative leverages the fact that most commercial BIOSes can be easily configured to boot from a CD if one is present at boot time. In this approach, the user performs a system scan by booting from a CD that immediately launches a copy of the security software. The security software downloads updates, if available, then scans the contents of the hard drive.

This approach offers several advantages. Since the BIOS is small and hard to modify [21], there is high assurance that the software security will be executed appropriately. The software itself loads from a CD, making it tamper-resistant as well. Also, the security software can execute before any other programs on the system begin, preventing malware from hiding in memory during the scan.

Unfortunately, this approach suffers from problems of efficiency and efficacy. First, as hard drives continue to increase in size, few users will have the patience to endure an entire drive scan every time the computer starts up. More importantly, scanning only at start-up means that the user has no real-time protection, even if new signatures are released. As the time between shutdowns continues to increase, users will spend more and more time in a potentially vulnerable state. Thus, while this approach offers a powerful alternative to virtualization-based approaches, its usability is limited.

4.3 Lessons Learned

Our exploration of these approaches provides two primary lessons. First, VM introspection cannot fully achieve the goals laid out in Section 2, since it cannot guarantee that the security software will receive accurate data if the guest OS is compromised. Second, protecting the security software by hardening the OS (as in the Watcher method) is excessively complex and largely infeasible for today's commodity operating systems. Thus, in our work, we leverage virtualization to isolate and protect the security software, but we inspect the guest VM at the file-system layer. By operating at a higher level, we avoid the complexity of inferring higher-level events based on low-level details, and we eliminate the insecure semantic gap engendered by the inference process. As a result, the security software receives accurate information about the software events inside the guest VM, even if malware executes and modifies the guest OS.

5 SAV-V Architecture

At a high level, the SAV-V architecture uses two VMs: a *secure VM* running the AV software and a *guest VM* running the original OS and applications. The SAV-V platform then offers the choice of two techniques,

Guest-Initiated Logging (GIL) and Split File System (Split FS), for ensuring the fidelity of the information needed by the AV software. Figure 1 summarizes these techniques. Finally, we leverage virtualization to create fake shutdowns that speed detection of patient malware. Below, we describe the architecture in detail.

5.1 Secure VM

By running the AV software in its own VM, SAV-V guarantees the integrity of its execution and the security of its connection to the outside world.

Separating the AV software from the legacy OS and applications makes maximal use of the isolation guarantees provided by the VMM. Any malware, malfunction, or misconfiguration in the guest VM cannot affect the operation of the security software in the secure VM. Like most virtualization-based work, we make the assumption that the vastly smaller VMM, both in terms of lines of code and interface exported, can be more readily trusted than the large commodity operating systems employed today.

Placing the security software in a separate VM also allows SAV-V to provide it with a guaranteed communication channel, since the VMM ultimately controls the requisite network hardware. Thus, security applications will always be able to download updates and new signatures, regardless of what happens inside the guest VM. Guaranteeing signature updates is an essential part of resilience against zero-day attacks. If the AV software cannot obtain the latest signatures, then the zero-day malware may remain undetected indefinitely.

5.2 Guest-Initiated Logging

In this section, we present the SAV-V technique of Guest-Initiated Logging (GIL). This technique allows AV software to detect disk-based malware even if it is initially unknown to the AV software and thus has an opportunity to execute. Disk-based malware, a class that includes most current viruses and trojans, is the traditional target of AV software. Even exploit-free systems remain vulnerable to disk-based malware, since social engineering attacks can convince users to execute malicious binaries. GIL does not attempt to detect zero-day mixed memory-based malware; for our solution to that problem, see section 5.3.

The key idea in GIL is to use an *append-only log* for file system writes. Before a write is allowed to proceed, it must be logged to an append-only log in the secure VM. Since disk-based malware must be written to disk before it can attain any control over the system, at the time it is written to disk it has no control over the system. Therefore, the guest OS will be uncorrupted at that time, and we can rely on it to initiate logging. By the time the malware takes control, the evidence of its presence will have already been irrevocably entered in the log. It will remain there until a matching signature becomes available, at which point AV software in the secure VM can detect it from its presence in the log.

We now describe how GIL works in greater detail. We add hooks to the guest OS that are activated by file system operations relevant to AV operations. The set of AV-relevant operations is quite small; most AV products only scan files on write, not read, and many other file system operations query or set attributes that are irrelevant to the AV scanner. Also, we only care about file system operations that succeed, so SAV-V need not log failed operations. Section 7 describes the full details of our implementation.

When one of the hooks in the guest OS is triggered, the SAV-V module invokes a VMM command that adds a log entry to an append-only log in the secure VM. The append-only property prevents a compromised guest OS from editing or deleting previous log entries. Even if malware in the guest OS deletes all of the incriminating files, their entries will remain in the log, allowing the AV software to detect the malware's presence when appropriate signatures are released.

Within the secure VM, the AV software monitors the log in realtime, allowing it to detect known viruses. When the AV downloads an update or a new set of signatures, it can travel through time by rescanning the log. Any viruses matched by the new signatures will be detected, even if they were unknown when they first

entered the system. To facilitate interoperability with legacy AV applications, SAV-V includes the ability to recreate files based on the log entries, so that legacy AV applications can perform their normal file-based scans without modification. With a standard log format, future AV products could include the ability to scan a log file *in situ*.

The reason GIL works on disk-based malware, but not memory-based malware, is as follows. A malware author aware of SAV-V and able to exploit a memory-based exploit could design malware that immediately disables GIL before writing to disk. More subtly, the malware could alter the SAV-V module in the guest OS to ignore files created by the malware while still logging normal file activity. This attack is not a problem when we restrict attackers to disk-based malware, since the malware's presence on disk will be logged before it has a chance to execute.

5.2.1 Discussion

In this section, we compare GIL with VM introspection. We also consider storage-space issues.

While GIL superficially resembles VM introspection, it differs significantly in several respects. Unlike VM introspection, GIL makes its dependence on the guest OS explicit. GIL expects the guest OS to log every malware-relevant file system operation. In contrast, VM introspection must make broad assumptions about the integrity of the guest OS; these assumptions are much harder to verify or enforce. In addition, allowing the guest OS to initiate logging makes the SAV-V system much simpler and more efficient than VM introspection, which must constantly monitor low-level hardware events and attempt to infer higher-level software events.

Users may be concerned about the amount of disk space consumed by the log in the secure VM. However, SAV-V permits the user to adjust the level of logging performed, trading off space efficiency versus ease of detection and recovery. Users can choose to log only the AV-related operations that have occurred since boot time, allowing SAV-V to determine if the computer is currently infected or it has been infected since the last boot. Alternatively, SAV-V can record more extensive logs that allow it to determine if the machine was ever infected and to potentially perform rollback and selective playback of legitimate actions [12, 16].

In practice, the capacity of consumer-grade hard drives continues to grow at a tremendous rate [37]. As disk sizes approach terabyte capacity, users are unlikely to mind (or even notice) if a fraction of the space is devoted to improving the security of their system. Furthermore, existing research indicates that full-file system logging can be performed on production servers [35] and that even the full-logging scenario can be implemented in a space-efficient manner [16, 20, 35].

Despite the growth in disk space capacity, there will always be some finite limit on the amount of space devoted to the SAV-V log. When SAV-V reaches that limit, the log must "wrap around", i.e., old entries will be overwritten with new entries. Malware might try to take advantage of this limitation by writing huge amounts of data to disk, in the hopes that its presence will be obliterated when the log is overwritten. These attempts can be countered via anomaly detection techniques. Assuming sufficient disk space is devoted to the log, wrap-around should occur infrequently. A sharp increase in the rate and amount of data written may signal an intrusion. SAV-V can warn the user that an intrusion may be underway, while noting that the warning may be unwarranted if the user is actively writing to the disk (e.g., by creating movies or downloading pictures).

5.3 Split File System

While GIL offers an efficient solution for monitoring file-system operations, it is unable to protect AV software against mixed memory-based malware. Thus, we designed an alternative scheme, called a Split File System (Split FS), which protects AV software against both disk-based and mixed memory-based malware.

As discussed in Section 5.2, GIL can be completely subverted by mixed memory-based malware. Once such malware takes control of the system via a memory-based exploit, it can prevent the guest from logging suspicious writes by the malware. Split FS, in contrast, guarantees that the security software in the secure VM sees every file-system operation with perfect fidelity. Thus, every file-system operation can be logged with perfect fidelity for review when updated signatures arrive. These guarantees are resilient to both known and unknown disk-based and mixed memory-based malware.

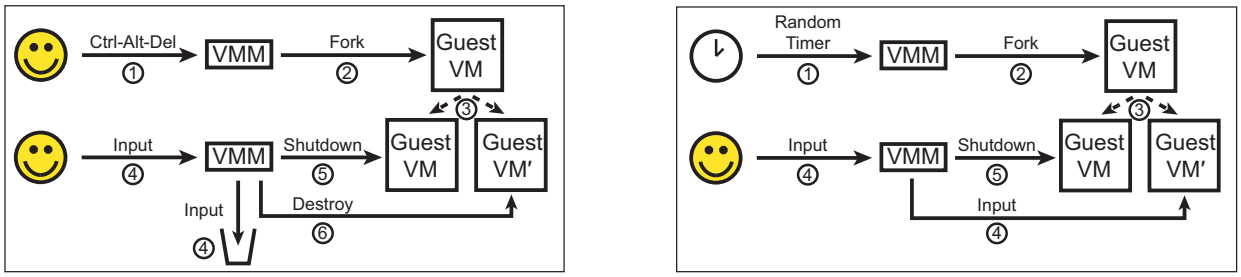
Split FS achieves its stronger guarantees by moving the file system itself to the secure VM and leaving only a stub interface in the guest VM, as shown in Figure 1(d). This way, the guest VM no longer has direct access to the file system or to the disk. Instead, it has a stub interface that communicates with the file system running in the secure VM. The stub interface it not trusted by the secure VM – it merely defines the set of commands the file system in the secure VM recognizes and facilitates the legacy OS's communication with the file system. The VMM facilitates message passing between the guest VM and the secure VM, taking advantage of the fact that both are operating on the same machine. The security software in the secure VM can monitor the messages arriving from the guest in realtime. These operations are also logged to facilitate review when siganture updates are received.

With the Split FS system in place, malware in the guest VM, regardless of whether it is disk-based or memory-based, must use the stub provided in order to write to disk. Thus, the security software and the logging software will always witness these operations with perfect fidelity. Since both operate in the secure VM, malware cannot interfere with their correct operation.

However, the interface between the guest VM and the secure VM must be carefully hardened against attack. Since we assume malware can compromise the guest OS, the interface on the secure VM must be prepared to handle arbitrary input from the stub in the guest VM. Thus, simply taking an existing file system and splitting it at an arbitrary point will likely leave the secure VM vulnerable to attack. Instead, the split must be made judiciously and the secure VM must thoroughly sanitize its inputs. Section 7 discusses how this consideration influenced our implementation.

As alluded to in Section 4.1.1, splitting the storage interface at the file-system layer (rather than, for example, at the block layer as Xen's blktap driver [2] does) has important security implications. Since AV software operates based on file-system events, making the split at the file-system layer ensures the accuracy of the AV's input. Splitting at any other level would require SAV-V to infer the requisite file-system events. For example, to make the split at the block level, SAV-V would need to correlate individual block reads and writes with file and directory accesses. While some researchers have considered the problem of inferring file system information from block-level events [30], their work focuses on performance optimizations and does not consider the possibility of an adversarial environment.

The Split FS technique can be seen as applying microkernel principles [27] to legacy operating systems. Indeed, previous work has suggested that VMMs are microkernels done right [17]. By separating the file system from the rest of the operating system, we gain strong guarantees about the integrity of the file system despite compromises to the rest of the guest operating system, but we still retain the benefits of legacy operating systems, particularly application compatibility. Some may argue that using VMMs in this manner negates some of the benefits of virtualization. After all, traditional virtualization makes each VM an independent unit that can be migrated, checkpointed, etc. independently. Making VMs interdependent as SAV-V does makes these properties harder to achieve. Nonetheless, the additional security benefits outweigh the complications entailed by mutually dependent VMs. Indeed, by modifying the VMM to include the notion of a container that holds one or more interdependent VMs, we can replicate the migration, checkpointing and other benefits of traditional virtualization while also enjoying the new security properties.



(a) **Real Shutdown** (b) **Fake Shutdown**

Figure 2: **Fake Shutdowns** *To create a convincing fake shutdown, we alter the normal shutdown procedure. During a real shutdown: 1) The user enters a unique key combination telling the VMM to shutdown; 2) The VMM sends a fork message to the Guest VM, which 3) creates a second Guest VM′. 4) Any further user input is discarded by the VMM. 5) The VMM uses a hardware signal to tell the original Guest VM to shutdown, and 6) destroys the Guest VM′. To fake a shutdown: 1) A random timer signals the VMM, which 2) sends a fork message to the Guest VM, which 3) creates a second Guest VM′. 4) Any further user input is redirected by the VMM to the new Guest VM′. 5) The VMM uses a hardware signal to tell the original Guest VM to shutdown. Notice that from the perspective of the original Guest VM, the two scenarios appear identical.*

5.4 Fake Shutdowns

The techniques we have described thus far all rely on disk-based scanning to detect malware. Unfortunately, patient malware may remain in memory until moments before the user shuts down the machine. Frequent shutdowns can hasten detection but disrupt the user. Instead, we propose a new technique for performing fake shutdowns that accelerates detection without disturbing the user. We also describe methods for ensuring that the fake shutdowns are indistinguishable from legitimate shutdowns.

5.4.1 Patient Malware

While our earlier techniques will quickly detect malware that writes to disk, an intelligent piece of malware that enters via a memory-based exploit and is aware of our system may avoid detection for extended periods of time by delaying any use of the disk.

Our techniques rely on disk-based scanning in large part due to the difficulty of scanning memory. Data in memory typically possesses less structure than that found in a file system. Reasoning about what structure is present requires assumptions about the guest OS (e.g., how it performs caching or assigns portions of the address space) that malware may violate. Portions of memory may also be paged to disk, further complicating any attempt to gain a coherent picture of memory contents.

Having gone through the effort to compromise a computer via a memory-based exploit, malware authors have an incentive to write to disk in order to persist across a shutdown[1]. However, a malware author may rationally choose to make the malware patient, i.e., she may design the malware to write to disk only when it detects that a shutdown is imminent. This way, if there is AV software running that can detect the malware by observing what it writes to disk, the malware can remain undetected for as long as possible. Furthermore, even if there is no AV software capable of detecting the suspicious disk data, the malware is nearly as persistent as otherwise. It will persist across a shutdown except in the unlikely event that the shutdown comes without warning, e.g., due to the power cord being abruptly pulled.

[1]The malware author may instead try to reinfect the machine using the same exploit. However, malware authors often compete with each other to collect compromised machines, so they have an incentive to patch the exploit they used initially to gain control. Thus, reinfection is no longer an option, and to maintain control, they must write information to disk

5.4.2 Frequent Shutdowns

The logical approach to force patient malware to write to disk is to frequently shut down the machine. Unfortunately, shutting down a machine is time-consuming and typically infrequent. Users dislike the disruption to their work and tend to delay shutting down as long as possible. Operating system trends suggest that we cannot wait for involuntary shutdowns either. As operating systems become more stable and require fewer patches, there will be fewer reasons to force the user to shut down the machine.

Instead, we propose employing virtualization to flush out patient malware via frequent shutdowns without disrupting the user. At a high level, we fork the state of the running guest VM. Existing work on VM migration [10] (i.e., moving a running VM from one physical machine to another) is directly applicable here. Indeed, this scenario is simpler, since it takes place on a single machine, without the overhead of network latency. In the foreground, the user can continue working on the original guest VM. In the background, the VMM can shutdown the forked copy. The security software in the secure VM can then scan any files written to disk during the shutdown process. Thus, when a machine shuts down, if malware fails to write to disk then it risks losing control of the system, and if it does write to disk then it risks detection.

5.4.3 Creating Convincing Shutdowns

For our fake shutdowns to work, they must be indistinguishable from real shutdowns. Any differences would allow the malware to ignore the fake shutdowns and wait for the legitimate ones. To provide this indistinguishability, we modify the shutdown procedure for the legitimate guest OS so that the fake shutdown can emulate it exactly.

To create convincing fake shutdowns, we modify the operating system running in the guest OS such that it will only shut down when it receives a particular hardware signal, for example, a signal on a particular interrupt line. When the user decides to shut down the guest OS, she invokes a canonical shutdown key sequence (e.g., Control-Alt-Delete). This key sequence is always captured by the VMM and never reaches the guest VM. The VMM prompts the user to confirm the shutdown (and warns her that all applications will be terminated). If the user cancels the shutdown, the VMM returns control to the guest OS. Otherwise, the VMM prevents further user input from reaching the guest OS. The VMM creates a checkpoint of the current state of the guest VM (using the same techniques as for a hot migration) and then sends the appropriate hardware signal to the guest VM, informing the guest OS that it should shut down. From the user's standpoint, the primary change (aside from the new key combination) is that unlike the current behavior of many operating systems, an application can no longer halt the shutdown. Thus, a user must be sure that all application state has been saved before confirming the shutdown.

With the modified version of the guest OS in place, the VMM can easily fake the shutdown of the forked VM. After performing the fork, the VMM simply sends the hardware shutdown signal to the forked VM. Since the guest VM never sees the key combination entered by the user, it cannot distinguish this shutdown from a legitimate user-initiated shutdown. Similarly, a machine performing the necessary migration steps to fork a VM for a fake shutdown will look identical to a machine preparing to shutdown.

Some care must be taken to insure that the fake shutdowns are performed on a pseudorandom schedule, or else malware may be able to infer whether a shutdown is fake based on when it occurs. Note, however, that even with random shutdowns, malware may try to distinguish a fake shutdown from a legitimate shutdown by analyzing the system's uptime and gambling that legitimate shutdowns happen only infrequently. Such malware risks losing control of the system if it inaccurately judges a shutdown to be fake, so it faces a difficult task in deciding when to write and when to wait.

6 Security Analysis

GIL (Section 5.2) targets disk-based malware. Since compromise by this kind of malware only occurs after malware is written to disk, GIL will have already logged the malware write by the time compromise occurs. Because the log is kept in the secure VM, out of the influence of malware running in the guest VM, attackers cannot tamper with the log. Furthermore, because the log is append-only, malware cannot delete the infected file to evade detection. Although GIL cannot be guaranteed to log memory-based malware (since the malware might disable GIL's logging infrastructure in the guest OS before writing to disk), it is useful against prevalent social engineering attacks that trick users into executing malicious code or data.

Split FS (Section 5.3) targets mixed memory-based malware in addition to disk-based malware. Split FS offers the property that as long as malware writes any data to disk, this data will be checked by AV software in the secure VM. Split FS, like GIL, uses append-only logging of file modifications to prevent zero-day malware from erasing any traces of its presence before signatures for it become available. Pure memory-based attacks are cleansed when the guest VM shuts down.

In the presence of Split FS, malware authors may realize that writing to disk exposes them to possible detection. Therefore, if they desire persistence, they may patiently delay writing to disk until it is absolutely necessary, i.e., when a shutdown is imminent. Our fake shutdown technique is designed to trick attackers into persisting to disk since the fake shutdowns cannot be distinguished from user-driven shutdowns. Some attackers may reason that the sooner a shutdown comes after the previous shutdown, the more likely it is to be fake. They may thus choose to gamble by persisting to disk only when the time between shutdowns is sufficiently large. However, attackers doing so risk losing control of the system if the shutdown is genuine, and provide all users, regardless of whether they use SAV-V, with a simple procedure for cleansing their system of the associated malware: reboot frequently.

Viruses may attempt to obfuscate or encrypt themselves to avoid detection by AV software. However, the role of SAV-V is to provide a secure platform for AV software rather than easing signature creation. The responsibility for discovering new malware and developing signatures remains with virus analysts.

Throughout this paper, we have made the assumption that the secure VM is more resistant to attack than the guest VM. This is because the only software it runs is the operating system and the security software we place in it. However, note that although the secure VM is more secure than the guest VM, it is not impervious to attack. The AV software, the file system exported by Split FS, SAV-V components such as the log replayer, and even the secure VM's operating system may have flaws that allow attack. Developers of the SAV-V code must be careful not to introduce so many additional flaws that protection is actually reduced by its presence.

Another possible avenue of attack is to exploit flaws in the virtual machine monitor itself. However, like most virtualization-based work, we make the assumption that the vastly smaller VMM, both in terms of lines of code and interface exported, can be more readily trusted than the large commodity operating systems employed today.

7 Implementation

In this section, we describe the details of the implementation and present the results of our evaluation. The evaluation considers both the effectiveness of the security of the system as well as the performance overhead it imposes. In evaluating performance, we do not consider the overhead of the virtualization itself, as this has been extensively examined in prior work [3, 6, 11]. Also, as hardware support for virtualization improves and becomes ubiquitous, we expect the overhead of virtualization to drop significantly.

7.1 Implementation Details

To evaluate SAV-V, we implemented the GIL and Split FS techniques. For the virtualization layer, we used Virtual PC, with Windows Server 2003 R1 as the host OS and Windows XP SP2 as a guest. To simplify development, we used the host as the secure VM.

Implementing GIL required changes to several system components. We wrote a file-system filter driver for the guest OS (Windows XP). The driver sits logically on top of the file-system driver and intercepts relevant file-system calls. We also modified the Virtual PC hypervisor to accept logging calls from the filter driver in the guest OS. When the filter driver intercepts an AV-relevant file-system operation, it invokes the logging call in the hypervisor, which then signals the Virtual PC application running in user mode in the host to append the operation to a log file. The file-system operations are logged synchronously on the guest side to ensure that disk-based malware files are always logged before they have a chance to execute, but the log entries are written asynchronously on the host side to improve performance.

The file-system calls we intercept are create, open, write, and close. We do not log creates and opens; we intercept them only so that we can initialize a data structure and associate it with the open handle for future use. This data structure includes the file's inode number and a bit indicating whether the handle has been written to. When we intercept a write, whether cached or non-cached, we log the write, including the bytes written and their offset. When we intercept a close, we log the close only if there has been a write to the handle.

The reason we log both cached and non-cached writes is as follows. We log cached writes because malware may be executed from the disk cache before it has been flushed to disk. By logging cached writes, we ensure we log such malware before it can take control of the machine and disrupt logging. We also log uncached writes because not all writes use the cache. Note that if malware is written to a file by memory-mapping that file, our scheme will not detect the cached write and will only log the write when it is flushed from the cache. This produces a short window of vulnerability for our scheme; fortunately, it only arises in the unlikely event that the user downloads the malware using a program that writes via memory mapping.

We also wrote a replayer application that interprets the log and recreates the relevant files. On each replayed close, it opens the file to make the host AV software's realtime scanner scan the modified file.

To evaluate the Split FS technique, we leveraged existing technology. On the host OS, we created a network-shared folder via Windows File Sharing. We connected the guest to the host using Virtual PC's virtual networking support and mapped the shared folder on the host as a network drive on the guest. Thus, all files are stored on the host, and the guest must access them via the Server Message Block (SMB) protocol. Since the SMB server interface is designed to be exposed to the world, it has been hardened against potentially malicious input, thus protecting the secure VM against a subverted guest VM. We also wrote a file-system filter driver that runs in the host to intercept and log relevant file-system operations on this shared folder. This filter driver is similar to the one we used in GIL, except that since it runs in the host, it does not need to signal the host to append to the log. It simply appends the log entry to a kernel buffer that is periodically and asynchronously appended to the log file by a worker thread.

While this implementation serves as a proof-of-concept, it is clearly not optimal from a performance standpoint. Messages passed between the guest and host must traverse the entire TCP/IP stack on both sides. This is particularly expensive on the guest side, since most networking-related operations must be emulated. Nonetheless, this implementation provides useful feedback on the amount of overhead imposed by the Split FS technique. The results can be seen as a worst-case scenario as there is ample room for improvement.

7.2 Detection

To demonstrate the effectiveness of GIL, we perform the following experiment. We save four files containing the BankAsh virus to the virtual hard drive of the guest VM, then delete those files. We do not have SAV-V perform online checking for viruses during this period, to emulate what would happen if this occurred when signatures were not yet available. Then, to emulate signatures becoming available, we replay the log of guest file-system activity on the host VM, and scan the resulting files with a virus scanner. The AV software raises an alarm, thereby detecting the past presence of the malware in the guest VM and thus the potentially corrupted current state of that VM.

7.3 Microbenchmark

To understand the performance impact of SAV-V on individual file operations, we developed a simple microbenchmark. This microbenchmark performs 100,000 trials, with each trial consisting of the following five operations:

1. Create a file.
2. Write a page (4 KB) of data to that file with caching disabled.
3. Read the page of data.
4. Close the file.
5. Write a page (4 KB) of data to that file with caching enabled.

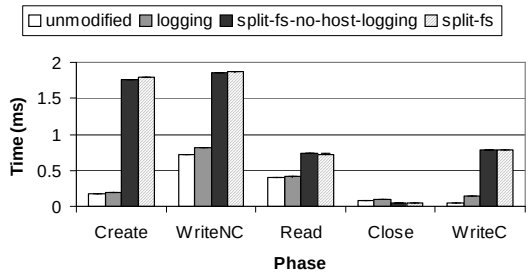
After each trial, the file is deleted.

We use the processor cycle counter to measure the duration of each operation, since these operations can be very short. We disable Virtual PC virtualization of the instruction to read the cycle counter so we can directly and efficiently read this hardware counter from the guest VM. Because our benchmark runs within a virtual machine, occasionally one of its operations is interrupted by the host operating system's scheduler to run a thread on the host, causing the operation to take substantially longer than it should. Because these events are rare but have a large effect on the mean, they can lead to misleading results. Therefore, we discard any operation taking longer than 10 ms, the granularity of the timer used for scheduling. This caused us to discard less than 1% of the trials for each operation.

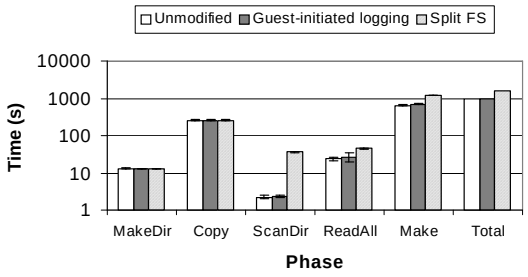
Figures 3(a) and 3(c) show results of this microbenchmark. Error bars in Figure 3(a) indicate 95% confidence intervals for the means. These bars are barely visible because the large number of trials gives us high confidence in the accuracy of our sample means.

For GIL, we see that the overhead is quite small. Average overhead is 0.02 ms for create, 0.09 ms for non-cached write, 0.01 ms for read, 0.02 ms for close, and 0.10 ms for cached write. This overhead increases operation completion time by 10% for create, 13% for non-cached write, 2% for read, 25% for close, and 196% for cached write. The create operation overhead is from intercepting the request and performing a query to obtain the file's inode number. The write overhead is larger because it involves communicating with the host virtual machine and transmitting the written bytes. As a percentage increase, this overhead is especially noticeable for the cached write, since the cached operation itself is quite short. The read operation overhead is from our filter driver intercepting the request, even though all we do is immediately pass on the request to the file system. Interestingly, the close overhead is also quite small. This is because it only reflects the time to intercept the request, not the time to log the close operation to the virtual machine. In Windows, the kernel-level close happens asynchronously after the user-level close is allowed to complete, so the time it takes to log this close operation is not observed by our benchmark.

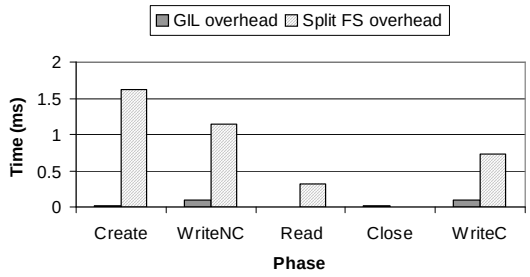
For Split FS, we see much higher overheads. Average overhead is 1.61 ms for create, 1.15 ms for non-cached write, 0.32 ms for read, -0.03 ms for close, and 0.74 ms for cached write. This overhead increases operation completion time by 891% for create, 160% for non-cached write, 80% for read, -33% for close, and 1490% for cached write. Interestingly, closes are faster for Split FS; this is because SMB caches handles to remote files even after they are closed, to save time in case they are opened locally later. So, less



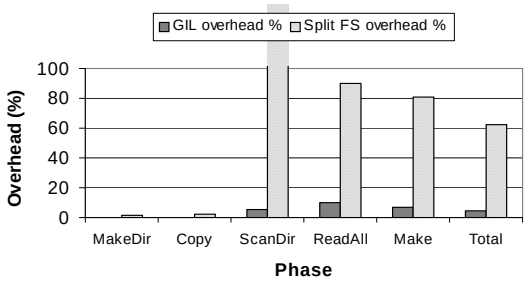
(a) **Microbenchmark Results**



(b) **Macrobenchmark Results**



(c) **Microbenchmark Results—Overhead**



(d) **Macrobenchmark Results—Overhead**

Figure 3: **SAV-V Inspection Techniques** *Figure 3(a) plots the mean time to perform each phase of the microbenchmark. Figure 3(b) plots the mean time to perform each phase of the MAB macrobenchmark on a logarithmic scale. Figures 3(c) and 3(d) plot the differences in means due to SAV-V overhead. WriteNC means non-cached file write, WriteC means cached file write, and GIL stands for Guest-Initiated Logging. Error bars indicate 95% confidence intervals. For the macrobenchmarks, Split FS employs logging.*

processing is performed than for a local file system, and there is no communication over the virtual network. On the other hand, all other operations require crossing the virtual machine boundary through the virtual network interface, and this causes a substantial reduction in performance. Most notably, reads suffer high overhead even though we do not log them, solely because they must use SMB over the virtual network.

By observing the difference between Split FS with and without host logging, we can see how much of the overhead of Split FS is due to host logging. We observe that host logging accounts for less than 0.03 ms per operation, i.e., only a small fraction of the total overhead of Split FS. We conclude that most of the overhead is due to the file-system operations over the network, and thus future optimizations that mitigate this aspect will have a substantial effect on the overall overhead.

7.4 Macrobenchmark

While microbenchmarks show us the overhead of individual file-system operations, they can produce an overly pessimistic view of the actual effect on users. This is because most applications do many things besides file system operations. Furthermore, many of the file-system operations are reads, which as we have seen, suffer little overhead in GIL.

To evaluate the application-level effects of SAV-V, we ran a Modified Andrew Benchmark (MAB). The Andrew Benchmark [18] consists of five phases of a compile job: In phase 1 (MakeDir), the benchmark creates an empty directory hierarchy. In phase 2 (Copy), the benchmark copies all of the source files into the new directory hierarchy. In phase 3 (ScanDir), it requests the status of every file. In phase 4 (ReadAll), it reads every byte of every source file. In phase 5 (Make), it runs the actual compilation. The modification

that makes our benchmark a MAB involves the source code used. The original Andrew Benchmark was developed in 1988 and targeted client computers with 65 MB disks, so the source code was only 200 KB in size. To provide a more realistic modern workload, and to make sure we at least forced the workload out of the processor's cache, we used the source code for Apache 2.2.3 for Windows. The source consists of 2,852 files and 202 directories for a total of 43 MB.

We ran this benchmark 100 times on each of our three setups: unmodified, GIL, and Split FS (with host logging). We report the mean duration of each phase for each setup. To avoid caching effects, we reboot both the guest VM and the physical host before each run. Furthermore, we employ differencing disks so that after each run, we can roll back any changes that were made to the virtual hard drive and start the next run with the same initial drive contents.

Figures 3(b) and 3(d) show the results. We see that GIL causes little overhead on this macrobenchmark. The difference between GIL and the baseline is statistically insignificant at the 95% confidence level for all phases except ReadAll; even for ReadAll, the overhead is only 10%. Considering all phases together, GIL increases completion time by only 5%.

Split FS, on the other hand, substantially increases the completion time of the benchmark. It increases completion time by 2% for MakeDir, 2% for Copy, 1508% for ScanDir, 90% for ReadAll, 81% for Make, and 62% overall. This total overhead is much greater than GIL because even for phases that involve no logging, such as ScanDir and Make, Split FS requires substantial overhead just to read files across the virtual network.

The overhead from Split FS is unsurprising, given that every file system operation must traverse the entire network stack on both sides. This is particularly expensive for the virtualized Guest OS, since most networking-related operations must be emulated. We believe a different communication strategy, optimized for the fact that the communicating virtual machines are colocated, could substantially reduce this overhead.

8 Future Work

As described in Section 5.2, GIL is vulnerable to memory-based malware. While Split FS closes this loophole, it does so at a great cost in performance. Therefore, we are working on techniques to detect mixed memory-based malware while maintaining similar performance characteristics to GIL.

One promising approach is to supplement GIL with periodic disk consistency checks. By checking whether the guest's virtual disk state is consistent with the set of file system operations the guest has logged, we may detect memory-based malware in the guest VM that attempts to halt or subvert guest-initiated logging. Thus, mixed memory-based malware will be detected by the consistency checks if it tries to subvert GIL, and by GIL itself if it does not.

One way to perform consistency checks is as follows. Periodically, SAV-V takes snapshots of the guest VM's virtual disk, doing so efficiently using copy-on-write disks. After each snapshot, it checks whether the latest snapshot is consistent with the sequence of file system operations logged during the period since the previous snapshot. To do this, it creates a virtual disk from the previous snapshot, replays the operations logged between that snapshot and the most recent one, and compares the resulting virtual disk with the most recent snapshot. If there is any inconsistency between the two at the file system level, this suggests that malware interfered with GIL. SAV-V can then alert the user that an intrusion is suspected.

For this scheme to work, we must log more extensive information than GIL does already. We must log any operation that modifies the file system, or else we may think the deviation between our replayed disk and the real disk signifies an intrusion. For instance, in addition to writes we also need to log creates, deletes, and size extensions.

This scheme also requires that the checker understand the correspondence between disk state and high-level file system state. This makes the checker complex and dependent on details of the file system im-

plementation. Errors in this code could lead to false positives, where we notify the user of a nonexistent intrusion, or false negatives, where we fail to notice a piece of malware that evades our faulty checks.

9 Related Work

IntroVirt [19] has a similar goal to SAV-V, aiming to detect past, even zero-day, exploits of vulnerabilities once the vulnerabilities become known. IntroVirt employs VM logging, rollback and replay [12] along with a predicate engine outside the guest VM to apply vulnerability predicates to the guest VM for detecting intrusions. Unlike our approach, IntroVirt takes a VM-introspection-based approach [15]. The authors of IntroVirt recognized the semantic gap in expressing vulnerability predicates using the low level, hardware abstraction exposed by the VM logging, and addressed it by allowing predicates to invoke code that already exists in the guest. In contrast, SAV-V's GIL performs logging at a much higher semantic level than IntroVirt, namely the file system level. Consequently, GIL logs significantly fewer events and yields much better performance, smaller log volume, and a significantly simpler log replay implementation. SAV-V offers easy deployability, since today's AV software can readily run on SAV-V; with IntroVirt, however, either the AV software must be modified to inspect virtual disks or IntroVirt needs to implement a complex VM introspection mechanism to infer file system operations on the guest. Lastly, SAV-V does not face some of the unsolved, difficult issues faced by IntroVirt, such as multi-processor support.

The NSA's NetTop platform [23] uses virtualization to allow the user to access information at different classification levels from a single desktop. It also includes a separate partition that filters out basic network attacks. SAV-V also uses virtualization to create a separate partition for security software, but SAV-V focuses on supporting anti-virus software, which involves considerably more interaction with the guest VM.

Pennington et al. explore storage-based intrusion detection systems [25], and their implementation places the intrusion detection system (IDS) on an NFS server. In a similar vein, Paul et al. propose to use the processor in modern disk drives to scan for viruses based on the I/O traffic seen by the disk drive [24]. SAV-V uses virtualization to place the security software on the same host machine, and it employs standard AV techniques and signatures, rather than an ad hoc IDS or specially crafted signatures for disk I/O. SAV-V also avoids the semantic gap engendered by inferring file-level operations from disk-level activity.

Many have leveraged the isolation feature of virtual machines for reliability [9, 13, 31] and security [14, 32]. Garfinkel et al used virtualization to develop a trusted computing platform on commodity hardware, allowing applications with varying security requirements to run in separate VMs [14]. Ta-Min et al propose a system that allows the programmer to partition an application's trust in the OS such that untrusted portions run on a commodity OS, while trusted portions run on a customized private OS in a separate VM [32]. While these systems use isolation to segregate trusted from untrusted software, SAV-V uses isolation to make protection software tamperproof. Isolation alone is not sufficient for SAV-V: we also require high fidelity inspection of the guest.

Several hardware systems have been proposed to improve the state of software security. CoPilot [26] uses a PCI add-in card to periodically check for malicious modifications to Linux kernels to detect rootkits. The Trusted Computing Group (TCG) is an organization that promotes open standards to help strengthen computing platforms against software-based attacks [1]. The TCG issued a specification for a Trusted Platform Module (TPM) [34], which is a dedicated security chip designed to enhance software security. With a secure boot architecture (such as AEGIS [4]), the boot process can be terminated if the software to be loaded at each stage fails to match known-good values stored in the TPM. Alternatively, the platform can perform a trusted boot by measuring each piece of software loaded, and storing these measurements in the TPM for later attestation to a third party. While it is possible to make an AV application's code tamperproof (or at least, tamper-evident) with CoPilot or the TPM-based boot processes, it is much harder to ensure the fidelity of all interactions between the AV software and the OS, as we argue in Section 4.1.2. The TPM-

based protections also provide most of their guarantees at boot time; if the computer is later infected, AV software can be compromised. Pioneer [29] can provide similar guarantees to a third party at arbitrary times after the computer boots, but it may be less suitable for AV applications that must constantly react to file system activity and typically run on consumer desktop machines, without a third-party verifier.

10 Conclusion

In this paper, we presented the design, implementation, and evaluation of SAV-V, a virtualization-based secure execution platform for AV software. SAV-V enables AV software to run with integrity and to detect past malware, even in the face of zero-day malware that compromises the operating system. We kept SAV-V simple and performant by avoiding complex VM introspection and OS-hardening techniques. Central to our approach is inspection of the guest VM at the file system level, matching the system semantics and the application interface that existing AV software expects. This also makes SAV-V practical for deployment.

References

[1] Trusted Computing Group. `http://www.trustedcomputinggroup.org/`, Mar. 2005.

[2] Blktap userspace tools + library. `http://lxr.xensource.com/lxr/source/tools/blktap/`, Aug. 2006.

[3] Advanced Micro Devices. AMD64 virtualization: Secure virtual machine architecture reference manual. AMD Publication no. 33047 rev. 3.01, May 2005.

[4] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, 1997.

[5] K. Asrigo, L. Litty, and D. Lie. Using VMM-based sensors to monitor honeypots. In *Proceedings of the ACM conference on Virtual Execution Environments (VEE)*, New York, NY, USA, 2006.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, Oct. 2003.

[7] CERT. Advisory CA-2003-20 W32/Blaster worm. `http://www.cert.org/advisories/CA-2003-20.html`.

[8] CERT. Advisory CA-2004-01. `http://www.cert.org/incident_notes/IN-2004-01.html`.

[9] Y. Chen, P. England, M. Peinado, and B. Willman. High assurance computing on open hardware architectures. Technical Report MSR-TR-2003-20, Microsoft Research, Mar. 2003.

[10] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, May 2005.

[11] I. Corporation. Intel virtualization technology specification for the IA-32 Intel architecture. Intel Publication no. C97063-002, Apr. 2005.

[12] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.

[13] U. Erlingsson, T. Roeder, and T. Wobber. Virtual environments for unreliable extensions. Technical Report MSR-TR-2005-82, Microsoft Research, June 2005.

[14] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[15] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *the Internet Society's Symposium on Network and Distributed System Security (NDSS)*, Feb. 2003.

[16] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara. The taser intrusion recovery system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, New York, NY, USA, 2005.

[17] S. Hand, A. Warfield, K. Fraser, and E. Kotsovinos. Are virtual machine monitors microkernels done right? In *Proceedings of the 10th USENIX Workshop on Hot Topics in Operating Systems (HotOS-X)*, Santa Fe, NM, June 2005.

[18] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, 1988.

[19] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[20] S. T. King and P. M. Chen. Backtracking intrusions. *ACM Transactions on Computer Systems*, Feb. 2005.

[21] P. Lang. Flash the Intel BIOS with confidence. *Intel Developer UPDATE Magazine*, Mar. 2002.

[22] Microsoft. Virtual PC. Available at: `http://www.microsoft.com/windows/virtualpc`.

[23] National Security Agency. Nettop: Technology profile fact sheet. `http://www.nsa.gov/techtrans/techt00011.cfm`.

[24] N. Paul, S. Gurumurthi, and D. Evans. Towards disk-level malware detection. In *Workshop on Code Based Software Security Assessments (CoBaSSA)*, Nov. 2005.

[25] A. Pennington, J. Strunk, J. Griffin, C. Soules, G. Goodson, and G. Ganger. Storage-based intrusion detection: Watching storage activity for suspicious behavior. In *USENIX Security Symposium*, Aug. 2003.

[26] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot—a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the USENIX Security Symposium*, 2004.

[27] R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, and M. Jones. Mach: A system software kernel. In *Proceedings of the Computer Society's International Conference COMPCON*, Feb. 1989.

[28] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. Browsershield: Vulnerability-driven filtering of dynamic html. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, Nov. 2006.

[29] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2005.

[30] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-smart disk systems. In *Proceedings of the Second USENIX Symposium on File and Storage Technologies (FAST '03)*, pages 73–88, San Francisco, California, Mar. 2003.

[31] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 22(4), Nov. 2004.

[32] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 2006.

[33] The SANS Institute. The top 20 most critical internet security vulnerabilities - press update. `http://www.sans.org/top20/2005/spring_2006_update.php`.

[34] Trusted Computing Group. Trusted platform module main specification, Part 1: Design principles, Part 2: TPM structures, Part 3: Commands. `http://www.trustedcomputinggroup.org`, Oct. 2003. Version 1.2, Revision 62.

[35] C. Verbowski, E. Kiciman, A. Kumar, B. Daniels, S. Lu, J. Lee, Y.-M. Wang, and R. Roussev. Flight data recorder: Monitoring persistent-state interactions to improve systems management. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, Nov. 2006.

[36] VMWare. VMWare Workstation. Available at: `http://www.vmware.com/`, Oct. 2005.

[37] C. Walter. Kryder's law. *Scientific American*, Aug. 2005.

[38] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of ACM SIGCOMM*, Portland, OR, Aug. 2004.