Transparent Detection of Computer Malware using Virtualization

Evan Broder

under the supervision of

Frans Kaashoek, Nickolai Zeldovich

MIT CSAIL

Abstract

In this paper, I explore malware detection using a combination of virtualization- and storage-based intrusion detection techniques. By monitoring disk activity of a virtual machine and correlating that activity to knowledge of the filesystem structure on the virtual machine’s disk, an intrusion detection system can react to file changes immediately. Such a system can use a traditional antivirus scanner from the virtual machine monitor on just those files modified within the virtual machine, avoiding the effect of rootkits and other mechanisms that can obscure the view of software operating within the virtual machine, while minimizing unnecessary scanning. I have implemented such a detection system by modifying KVM for Linux, and have used it to observe and scan a Windows XP virtual machine with a FAT32 filesystem. The system was able to efficiently detect malware infections.
1 Introduction

Viruses, worms, and other malware are an omnipresent part of today’s online experience. Preventing the spread of such malware is an important task for systems administrators. There’s been extensive research into an assortment of techniques for malware detection, taking advantage of semantic separation to transparently monitor infected machines and their behavior from outside the malware’s sphere of influence.

In a popular web comic [12] (reproduced in Figure 1), the character builds a “Malware Aquarium,” which displaying a series of virtual machines that are randomly created and destroyed. Each virtual machine uses activities traditionally believed to attract malware (such as opening e-mail attachments or clicking on pop-up links) to try and actively acquire malicious software. The display shows not only the virtual machines, but the malware that they have acquired as it spreads throughout the virtual network.

Because we feel the “Malware Aquarium” represents an innovative way to visualize and analyze malware’s effect over time in a network of computers, we have attempted to recreate portions of the “Aquarium.” The Malware Aquarium project is an ongoing group effort,
with many students working on different aspects of the aquarium.

In order to make the display engaging, it is important to quickly and accurately detect new malware with good enough performance to scale to large numbers of virtual machines. I have created such a detection system for the Malware Aquarium. My detection system can accurately detect malware quickly after it is written to the disk of a virtual machine. While I implemented the detection system using KVM running Windows XP virtual machines on a FAT32 filesystem, my techniques are not specific to either the virtual machine monitor or the virtual machine operating system and filesystem being used.

Section 2 explores the approaches of virtualization- and storage-based intrusion detection and justifies using those approaches for accurate malware directions. Section 3 describes the environment both of the virtual machines and the virtual machine monitor. Section 4 examines the various tasks needed to implement my detection system, as well as how I implemented those tasks. Section 5 evaluates the performance of our intrusion detection system. I close with a discussion of related work and conclusion (sections 6 and 7).

2 Detecting Malware

Today’s malware has moved past simply compromising and using a victim’s resources. With the spread and popularity of antivirus software, viruses and worms are constantly engaged in a game of cat and mouse with intrusion detection systems.

Instead of simply modifying the files and programs on the filesystem to effect the desired malicious behavior, malware attempted to “get under” antivirus software using “rootkits.” Rootkits can modify the core operating system, generally to hide files, processes, and other resources from the view of programs outside the operating system. In this way, they can escape even being seen by traditional userspace applications.

As malware and intrusion detection continue to fight to see who can control the lowest
level, researchers, such as King et al. in [8], have theorized that the next step for malware will be to install itself as a virtual machine monitor, existing outside the view of the operating system itself.

2.1 Detection with Virtualization

In order to protect intrusion detection systems against such powerful malware, researchers have sought to remove the detection systems from the machine under inspection. Some of these isolated intrusion detection systems work by monitoring network traffic [4, 11, 16], others by monitoring storage [13]. One popular technique is using a virtual machine monitor and analyzing the actions of a virtual machine running under that monitor [1, 3, 5, 6].

Because of the “safety” property of a virtual machine monitor [14], analysis of a virtual machine from the monitor gives the guarantee that no compromise or modification of the virtual machine’s operating system can alter the virtual machine monitor’s ability to observe actions taken by the virtual machine. Additionally, because all of the resources of the virtual machine are under the control of the virtual machine monitor (again due to the safety property), the monitor is guaranteed the ability to inspect all actions taken by the virtual machine. I opted to implement our intrusion detection system in the virtual machine monitor for these reasons.

2.2 Detection through Storage

Given the virtual machine monitor’s ability to inspect all actions of the virtual machine being monitored, I needed to choose which actions to monitor. In searching for malware, I made the assumption that malware authors desire a persistent presence on their victims’ computers, i.e. they want the malware to remain on the targeted machine across reboots so that they can continue conducting malicious activity. Such malware could only be persistent
if it is written to the hard disk, since all other state is transient.

When designing their storage-based intrusion detection system, [13] examined eighteen common malware toolkits. Of those, only three made no changes to the persistent storage of the machine under test. Furthermore, Pennington et al. were studying malware toolkits, not malware, and they suggested that consumers of the toolkits would use their own mechanisms to make the rootkits’ effects permanent. Since I assumed that all malware writes to the filesystem, I selected monitoring the filesystem as the most foolproof way to detect malware infection.

2.3 Challenges

While combining virtualization- and storage-based intrusion detection methods gives the detection system a powerful advantage over malware, these techniques present some challenges in implementation. Because my intrusion detection operates outside of the virtual machine being inspected, it is unable to take advantage of the abstractions normally provided by the operating system. This means that in order to examine concepts such as “files,” “processes,” or “network streams” from the intrusion detection system, they must be assembled from their components of “hard-drive sectors,” “memory pages,” and “packets” separately from the operating system. However, this separation, or “semantic gap” [1], proves to be desirable, because I wish for my detection system to see through compromises of the monitored operating system, and thus compromises of the abstractions (such as “files”) provided by that operating system.

3 Monitoring Environment

With the Malware Aquarium project, our goal was to create a compelling and engaging display of malware spreading through a network of virtual machines. Given its large number
of well-known security vulnerabilities, we opted to use Windows XP without any service packs or other post-release updates for our virtual machines. It was our hope that this configuration would present a highly vulnerable target for malware. To simplify the code necessary to bridge the “semantic gap,” we opted to use the FAT32 [10] filesystem in the virtual machines.

To manage the virtual machines, we selected KVM [7], managed using libvirt [9].

4 Design of Intrusion Detection System

The goal of my intrusion detection system was to identify malicious files as quickly as possible after they are written to the filesystem, since I assumed that all malware will be written to the filesystem. However, because my detection system is running from the context of the virtual machine monitor, and therefore exists in the wrong side of the “semantic gap,” the only information available to the detection system is which sectors on the virtual machine’s disk image are written to.

In order to accurately and efficiently detect malware, the intrusion detection system needed to be capable of efficiently completing the following tasks:

1. Receiving a notification when the virtual machine writes to its disk, including the location on disk that was written.

2. Mapping from a write to a particular location on the virtual machine’s disk image to the file in the virtual machine’s filesystem that was modified

3. Extracting a particular file in the virtual machine’s filesystem to the virtual machine monitor for examination by traditional antivirus software.

4. Maintaining the mapping from block device to file name in the face of ongoing writes to the filesystem.
The intrusion detection system itself was implemented as a separate program from the 
virtual machine monitor. However, the first task required modifying the virtual machine 
monitor as well. I examine our approach for each of these tasks.

4.1 Write Notification

Because the virtual machine monitor is responsible for actually executing reads from and 
writes to the disk on behalf of the virtual machine, I chose to modify the virtual machine 
monitor to notify our detection system every time a write to disk occurred.

Our virtual machine monitor, KVM, uses the open-source machine emulator QEMU [15] 
to emulate devices connected to the virtual machine, including the virtual machine’s hard 
drive. I modified each of the mechanisms through which QEMU writes to a virtual block 
device backend. Under my modifications, every write causes QEMU to send a 16-byte 
datagram over a UNIX domain socket containing the offset and length of the write that just 
occurred as 64-bit (i.e. 8-byte) unsigned integers.

These modifications proved to be straightforward; they required adding 64 lines of C 
code across 4 files.

4.2 Block-to-File Translation

When my detection system receives a notification of a disk write operation from QEMU, 
the system needs to use the information of which location on disk was written to in order to 
determine which file was modified.

A FAT32 filesystem can be broken into three regions: the reserved region, the File Al-
location Table region, and the file and directory data region [10]. The reserved region is 
not used by the FAT32 filesystem at all, and the File Allocation Table region, while used to 
connect segments of data holding logically continuous information, does not hold any actual
\[
\text{cluster\_number} = \frac{\text{offset} - \text{cluster\_start\_offset}}{\text{bytes\_per\_cluster}} + 2
\]

Figure 2: Formula for converting an offset from the beginning of a disk image to a cluster number. N.B.: Cluster indexes begin counting at 2, not 0.

filesystem data. The contents of all files and listings of all directories are stored in the data region. Thus, any write outside of the data region can be ignored for the purposes of finding modified files that need to be scanned.

The data region is broken into “clusters” of between 1 and 128 contiguous hard disk sectors, similar to the “block size” on filesystems such as ext2 or ext3. In order to translate a particular sector on the hard disk into a particular path in the filesystem, the detection system maintains a mapping from a location on the virtual machine’s disk to the path in the filesystem whose data is contained in that location. Because an entire cluster can only contain data for a single file, the mapping the detection system maintains uses the cluster index for a key.

Thus, when the detection system is notified of a disk write, all it needs to do is convert the sector offset into a cluster index (see Figure 2), and use that cluster index as a key in the cluster-to-file mapping. This uses memory proportional to the number of allocated clusters in the filesystem being monitored. However, it allows for translating the offset of the disk write into a filename in a constant amount of time.

In order to initially populate the mapping, the detection system performs a search over every file and directory in the filesystem, and for each file or directory, the system determines which clusters contain that file or directory’s data. In a FAT32 filesystem, the File Allocation Table serves as a linked list connecting clusters holding logically continuous data. Because of this linked-list-style structure, finding all clusters containing data for a particular file requires time proportional to the number of clusters used by that file. Thus, searching the
entire filesystem to initially populate the cluster-to-file mapping requires time proportional to the total number of clusters in use in the filesystem.

While I have only written code to generate and utilize such a mapping for the FAT32 filesystem, the concept of correspondence between filesystem entries and locations on the physical disk is common to all filesystems. Thus, while the details of the mapping might vary, this approach to interpreting disk writes is applicable to all filesystems. Furthermore, although the multiplicative constant factors may vary, the performance properties of such a mapping should remain the same for any filesystem.

4.3 File Extraction and Scanning

Once the file modified by the virtual machine has been identified through the cluster-to-file map described in subsection 4.2, the next step is to extract that file from the virtual machine’s disk image and copy it into the virtual machine monitor’s file system for inspection.

I opted against using Linux’s built-in support for FAT32 when extracting files. Linux’s FAT32 support is unable to cope with the ongoing disk writes from the virtual machine. By developing my own support for file extraction, I was able to avoid caching effects and ensure that I always extracted the file as it existed in the disk image.

Once the modified file has been copied into the virtual machine monitor’s filesystem, it can then be examined by any number of means. Because the focus of my detection system was on the monitoring of changes in the filesystem, I opted to use a turnkey solution for inspecting extrated files. My detection system uses Clam AntiVirus [2] to determine whether or not a particular file should be classified as malware.

The extraction of a file requires time and space proportional to the length of the file. Because the Windows XP boot partition contains several multi-gigabyte files used by the operating system as extentions of RAM (e.g. PAGEFILE.SYS, used for RAM swap space; and hiberfil.sys, used for RAM-to-disk system hibernation), I opted to only extract and scan
files under a certain size (1 gigabyte). I believe that this limitation could be overcome by using the information already available about the location of writes to only extract those parts of these massive files which had changed.

As an additional performance optimization, I addressed the case of a single file being written to repeatedly (or being written to in different sections). For each file that the intrusion detection system determines to have been modified, that file is only extracted when a different file is written to. Since I have assumed that malware’s goal is to create a persistent presence on the victim’s machine, this technique does not compromise the detection system’s ability to catch malware, since all modified files are eventually scanned. Observationally, this does not seem to have been entirely sufficient, as we frequently observed two files being alternatingly written to. I believe that a substantial performance improvement could be gained were the detection system to account for this type of usage pattern.

4.4 Maintaining Block-to-File Map

Over time, files are created, deleted, moved, lengthened, shortened, and otherwise changed. It is possible that any of these modifications could affect the cluster-to-file mapping developed in subsection 4.2. Therefore, as the intrusion detection system receives information about writes, it must update the cluster-to-file mapping.

In order to maintain this mapping, the intrusion detection system must maintain two other data structures in addition to the mapping itself, both of which effectively cache information encoded in the filesystem in order to efficiently determine how the filesystem changed.

The first supplementary data structure is a mapping from a path in the virtual machine’s filesystem to the list of clusters containing that path’s data. This structure is effectively the inverse of the cluster-to-file mapping, and allows the intrusion detection system to notice when a cluster previously allocated for a particular file is no longer used by that file.
The second data structure contains the entries of each directory on the filesystem, as well as the file metadata encoded in the directories on disk in a FAT32 filesystem. This allows the system to easily observe when files are added to or removed from a directory. Because a file’s stat information changes when its size changes, this also gives us an additional notification when a file grows enough to require a new cluster allocation.

For disk writes to locations containing files (i.e. not directories), no work is needed to maintain the mappings. For disk writes to directories, the system compares the cached directory entries against the directory entries in the filesystem. Any directory entry that does not match the cached entry should be examined to determine if it is out of date. The algorithm is described in detail in Figure 3.

Note that, update_cached_info_for_path and remove_tree_from_caches, the two helper functions used in Figure 3, update the stat_cache, cluster_to_file, and file_to_cluster data structures. The last of these, file_to_cluster, is used only by those helper functions to clean up previously cached values.

5 Performance Evaluation

In a practical deployment of this intrusion detection system, the system would likely compete with the virtual machine being inspected for resources from the virtual machine monitor. The two resources that could have a significant impact on the performance of the virtual machine are CPU utilization and RAM utilization. I examine each of these in turn.

All performance measurements and testing were conducted on a fast, multicore server with an abundance of memory. Virtual machines were allocated a single virtual CPU (i.e. a single core) and 1024 MB of memory. During my testing, the detection system did not compete with the virtual machine for resources.
def process_write(offset):
    if offset < cluster_region_start:
        return

    cluster = offset_to_cluster(offset)
    if cluster not in cluster_to_file:
        return

    queue = Queue()
    queue.push(cluster_to_file[cluster])
    modified = []

    while not queue.empty():
        path = queue.pop()

        if not isdir(path):
            modified.append(path)
        else:
            # For directories that have been written to, compare the
            # the cached directory entries and the current entries
            for filename in list_directory(path):
                if filename not in stat_cache[path] or 
                    stat_cache[path][filename] != stat(join(path, filename)):
                    # Update the file_to_cluster, cluster_to_file, and
                    # stat_cache entries for the file
                    update_cached_info_for_path(join(path, filename))
                    queue.push(join(path, filename))

            # Remove from all caches any files which are no longer
            # present in the directory
            missing_files = set(stat_cache[path].keys())
            missing_files -= set(list_directory(path))
            for filename in missing:
                # Remove the file and all its children from our cached
                # information
                remove_tree_from_caches(join(path, filename))

    return modified

Figure 3: Algorithm for processing the effects of a single disk write. The `process_write` function updates all data structures affected by a disk write to either a file or a directory, and returns a list of all files which are believed to have been modified.
5.1 CPU Utilization

Because my detection system sleeps between write notifications from QEMU, the system incurs no CPU overhead when no disk writes are occurring. When disk writes do occur, the CPU overhead is proportional to the frequency of writes.

To evaluate this overhead, I started a virtual machine, and while running under my detection system, I spent several minutes browsing a multimedia-heavy website (techcrunch.com) in Internet Explorer. I then shut down the VM. Such a browsing session is very write intensive, due to browser caching.

Over the course of the 517 second long browsing session, QEMU logged 10.6 megabytes worth of writes (21,616 sector-length writes). As a result of those writes, my intrusion detection system extracted and scanned 10,377 files.

During that browsing session, the intrusion detection system was utilizing CPU approximately 29.4% of the time, split nearly evenly between userspace computation and kernelspace privileged operations (almost exclusively calls to lseek(2) and read(2)). Figure 4 shows the breakdown of CPU utilization by the intrusion detection system.

With a total of 152.11 seconds not spent in an idle state, the intrusion detection system required approximately 7.0 milliseconds of CPU time for each sector (512 bytes) written to the virtual machine’s disk. Alternatively, the system required approximately 14.7 milliseconds for each file that needed to be extracted and scanned.

5.2 RAM Utilization

There are several structures used by the intrusion detection system to quickly map block writes to modified files. Some of these structures require memory proportional to the number of files in the filesystem; the rest require memory proportional to the number of blocks that have been allocated for file and directory data.
### Table 1: RAM used by the intrusion detection system for data structures for several different VMs with different size filesystems

<table>
<thead>
<tr>
<th>VM</th>
<th>Files and directories</th>
<th>Allocated clusters</th>
<th>Data segment size (kB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM 1</td>
<td>14,578</td>
<td>555,185</td>
<td>96,500</td>
</tr>
<tr>
<td>VM 2</td>
<td>15,547</td>
<td>360,847</td>
<td>74,584</td>
</tr>
<tr>
<td>VM 3</td>
<td>23,192</td>
<td>748,554</td>
<td>148,700</td>
</tr>
</tbody>
</table>

To measure the memory requirements for my intrusion detection system, I took three Windows XP virtual machines using the FAT32 filesystem and populated the data structures for my detection system using their disk images. I examined the size of the data segment for the process containing the data structures. Table 1 shows the parameters for each of the virtual machines sampled.

Based on correlating these three samples, the intrusion detection system uses approximately 3.2 kB of memory per file or directory in the filesystem under inspection. It additionally uses approximately 0.13 kB of memory per cluster on the filesystem that contains data.
6 Related Work

There has been extensive research conducted into detecting malware. I examine the relationship of my intrusion detection system to these.

Pennington et al. implemented an NFS-based intrusion detection system [13], which detected malicious activity at the networked storage layer. They used a series of predicates applied to specific paths to heuristically detect suspicious behavior. In spite of detecting suspicious behavior from outside of the context of the victim, monitoring from within a network-attached storage solution allowed them to operate with knowledge of filesystem semantics. However, while their techniques were limited to systems using a networked filesystem where the fileserver had filesystem awareness. My intrusion detection system was not designed to use heuristic methods to identify malicious activity. However, my system can in principle monitor arbitrary operating systems and arbitrary filesystems, and does not require that the machine being monitored use networked storage.

The “Stealthy Malware Detection” system [5] created by Jiang et al. is more similar to mine. They use knowledge of the filesystem and process table to examine state from the virtual machine monitor. They then compare this information to the same information, gathered from within the virtual machine. Discrepancies between the observed state inside and outside of the virtual machine can be identified as malicious activity. Additionally, they use traditional antivirus software to examine the entire filesystem from the virtual machine monitor. However, their detection-via-discrepancy only allows them to detect malware that uses rootkit techniques to mask its presence, and both detection techniques require on-demand scanning. By contrast, my intrusion detection system monitors a target continuously, and can quickly detect malware at the moment of infection.
7 Conclusion

I have presented a new virtualization- and storage-based intrusion detection system, capable of dynamically, efficiently, and quickly detecting malware as it is written to the disk of a virtual machine. While my intrusion detection system has only been implemented for the KVM virtual machine monitor and target virtual machines with FAT32 filesystems, the techniques used are applicable to all virtual machine monitors and target filesystems. Even with numerous known performance deficiencies, the detection system imposed only a small overhead. With performance improvements, I believe that such a system could effectively avoid the weaknesses of a traditional in situ antivirus solution.

8 Acknowledgments

This work was conducted in collaboration with Frans Kaashoek and Nickolai Zeldovich in the Parallel and Distributed Operating Systems group at CSAIL and the Student Information Processing Board. Other students working on pieces of this research include Quentin Smith, Jiawen Chen, David Benjamin, and Alan Huang.

References


