SEER: Practical Memory Virus Scanning as a Service

Jason Gionta  
NC State University  
jgionta@ncsu.edu

Ahmed Azab  
Samsung Electronics Co., Ltd.  
ahmedmoneeb@gmail.com

Peng Ning  
NC State University  
pningncsu.edu

William Enck  
NC State University  
whenck@ncsu.edu

Xiaolan Zhang  
Google Inc.  
czhang.us@gmail.com

ABSTRACT

Virus Scanning-as-a-Service (VSaaS) has emerged as a popular security solution for virtual cloud environments. However, existing approaches fail to scan guest memory, which can contain an emerging class of Memory-only Malware. While several host-based memory scanners are available, they are computationally less practical for cloud environments. This paper proposes SEER as an architecture for enabling Memory VSaaS for virtualized environments. SEER leverages cloud resources and technologies to consolidate and aggregate virus scanning activities to efficiently detect malware residing in memory. Specifically, SEER combines fast memory snapshotting and computation deduplication to provide practical and efficient off-host memory virus scanning. We evaluate SEER and demonstrate up to an 87% reduction in data size that must be scanned and up to 72% savings in overall scan time, compared to naively applying file-based scanning approaches. Furthermore, SEER provides a 50% reduction in scan time when using a warm cache. In doing so, SEER provides a practical solution for cloud vendors to transparently and periodically scan virtual machine memory for malware.

1. INTRODUCTION

Virus Scanning-as-a-Service (VSaaS) has become a popular topic for research and industry [4, 9, 17, 20, 25]. However, existing solutions only scan secondary storage. As malware continues to mature, file-only virus scanning is insufficient. We have already seen several instances of “Memory-Only Malware” that evades detection by many popular virus scanners [19, 14, 15]. In fact, only 30% of the most popular host-based virus scanners support memory virus scanning [16], and the computational overhead of their signature matching techniques makes memory scanning an infrequent event.

Memory-based VSaaS would provide benefit to both cloud clients and providers as Virtual Machines (VMs), like traditional hardware environments, require proper security measures to minimize threats and detect intrusions. However, designing a practical VM memory scanning service is non-trivial as (1) virus scanning is traditionally a CPU intensive task that can negatively impact VM operations, (2) memory that changes during analysis can lead to false positives (2), (3) scanning multiple VMs concurrently can impact VM and host operations (i.e., “anti-virus storms”) [23], and (4) file scanning optimizations are less effective for memory.

In this paper, we propose SEER as an architecture alternative to in-host memory scanning that enables efficient, transparent Memory VSaaS for virtualized cloud infrastructures. SEER “peers” into running VMs to efficiently identify malware in resident memory. SEER is built on DACSA, a decoupled architecture for cloud security analysis, to minimize guest and host performance [7]. DACSA enables fast memory snapshots and memory extraction. SEER processes snapshot data off-host to identify similar memory and duplicate scanning computation thus leading to a reduction in scan time. In doing so, SEER allows cloud providers to scan guest memory much more frequently and efficiently than naively applying existing file based scanning approaches.

Computation deduplication in SEER is achieved using a new algorithm that we call Prefix Based Scanning. Prefix Based Scanning is optimized by identifying partial data similarities across all memory segments to scan and normalizing similar data. For example, many Windows processes load ieframe.dll each containing small differences based on load address and runtime state. As a result, Prefix Based Scanning reduces the amount of computation to scan by only scanning identical prefixes once. Furthermore, our Prefix Based Scanning algorithm also identifies memory pages not present and virtually “fills in” similar memory from other processes per VM to increase virus scanner accuracy.

As a proof of concept to analyze the efficiency of SEER, we adapt the popular open source virus scanner ClamAV [10] to enable Memory VSaaS with Prefix Based Scanning. Our results show that SEER is practical and can be deployed by cloud vendors to find both existing file based malware and stealthy Memory-Only Malware in memory.

We make the following contributions:

- We propose a new architecture for efficient Memory VSaaS. SEER addresses a major gap present in existing Virus Scanning-as-a-Service solutions. We describe and overcome the challenges of memory scanning that prevent naive application of traditional file based scan-
ning techniques.

- We propose Prefix Based Scanning for efficiently scanning dynamic memory. Our proposed approach works to consolidate data thus reducing the amount of computation required to scan memory. The approach can be adopted by existing host-based memory virus scanners to reduce scanning computation.

- We provide an experimental evaluation of SEER and show our design reduces the data size to be scanned up to 87% and the size to scan up to 72%, compared to naive scanning approaches. We also demonstrate that SEER can correctly identify memory malware in infected machines.

The remainder of this paper proceeds as follows. Section 2 provides background on Memory-Only Malware and the challenges faced by memory scanning. Section 3 contains our design, implementation, assumptions and threat model. Section 4 evaluates SEER. Section 5 discusses deployment, security and privacy concerns, and limitations. Section 6 discusses related work. Section 7 concludes.

2. MOTIVATION AND BACKGROUND

2.1 Memory-Only Malware

Memory-Only Malware (MoM) identifies a class of stealthy malicious software that does not modify files on a target system. As such, MoM bypasses file-based detection techniques and the majority of popular virus scanners [16]. MoM trades persistence for stealth. While MoM will not survive system reboots, attackers are using new ways to gain persistence through “Watering Hole Attacks” [18].

MoM is not a new concept. The infamous Code Red Worm propagates only through memory [19]. More recently, security analysts have identified a Hydraq/McRAT variant that allows remote access to infected machines [15]. Finally, the popular MetaSploit framework provides an “out-of-the-box” MoM called Meterpreter [14].

Malware such as Zeus and Conficker also benefit from the lack of memory scanning. They only need to bypass virus scanning software once to gain a foothold. Once initial detection is bypassed, the malware uses code injection to inject benign active processes [12]. Injected code allows malware to stay resident in the running machine even when physical files are later found and removed. As a result, malware analysts recommend using memory dumps to discover and identify malware [12].

2.2 Scanning Memory

Memory scanning consists of analyzing virtual memory to find malware based on virus definitions. Previous work [9] has applied file-based scanners to physical memory. However, scanning physical memory will lead to incorrect results because virus definitions require correct context to match definitions. As a result, a memory-based virus scanner must scan virtual memory to ensure correct context. For modern operating systems, each process has its own virtual memory.

The virtual memory of a process \( p \) is a set of memory segments \( S_p \). Each memory segment \( s \in S_p \) represents an individual memory allocation managed by the operating system (OS) with size \( s_{\text{size}} \) bytes. Memory segments can contain stacks, heaps, or mapped files. Then \( p_{\text{size}} = \sum s_{\text{size}}, \forall s \in S_p \) represents the size of memory allocated for process \( p \). Malware can potentially reside in any process memory segment, therefore all processes on the OS, represented as the set \( P \), must be scanned. That is, \( p_{\text{size}} = \sum p_{\text{size}}, \forall p \in P \) bytes must be scanned.

A naive approach would be to scan all \( P_{\text{size}} \) bytes of memory. However, this faces the following challenges:

C1. Scanning is computationally intensive requiring significant CPU resources to match signatures. Scanning often disrupts critical operations during analysis. A large \( P_{\text{size}} \) can result in minutes of scanning and thus is not ideal for production environments.

C2. Memory modified during analysis can impact the correctness of scanner results, leading to false positives and negatives [2]. This occurs because scanners must copy, examine, and analyze memory that is constantly changing. As a result, false assumptions are made about the consistency of the data being analyzed.

C3. Multiple VMs scanned at the same time on a single host leads to “Anti-Virus Storms”, which can affect non-scanning guests and host operations [25].

C4. Hashing is widely used by file-based virus scanners as an optimization to limit the amount of data to scan. Memory is often unique which limits the effectiveness of hashing memory segments to identify previously scanned data. We find that previous hashing techniques used for file-based scanning [20, 17] only reduces the memory segments to scan by \( \approx 14\% \) for a single VM.

3. SEER

Next we give an overview of the SEER architecture. We discuss the disparate components for enabling SEER and the workflow for scanning VMs. We then detail specific design decisions for Prefix Match Scanning.

3.1 Overview

The SEER architecture enables practical, VM transparent, efficient memory scanning for virtualized cloud infrastructures. SEER overcomes challenges (C1-C4) by shifting memory scanning to a third party in which (1) guest VM memory is extracted and only unique data is shipped off-host, (2) a scan scheduler pre-processes data to identify similarities in data to scan, and (3) a modified scanner uses pre-processed meta-data to identify viruses in an isolated environment. The VM and host only incurs a small cost for extracting and shipping data (C1,C3) and prevents malware or exploits that targets the scanning process [17].

Figure 1 illustrates the main components and flow of SEER. The cloud provider will request/schedule (1) VMs for scanning. The scan scheduler initiates a scan (2) by requesting the memory extractor to extract memory and then ship memory meta-data and unique page data for pre-processing. The scheduler (3) stores the data and builds a scan configuration based on data extracted. The scan scheduler then triggers a new scan (4) using the generated scan configuration and (5) results are relayed to the cloud provider. For our design discussion, we consider QEMU/KVM as the host and Windows as the guest VM operating system.

On the virtualization hosts, we leverage DACSA to support fast, non-intrusive VM memory snapshots and memory
Figure 1: SEER Overview

extraction. DACSA creates a logical static copy of memory in milliseconds without copying data byte-by-byte [7]. This prevents consistency issues which can arise from analyzing memory that continues to change (C2). Memory segments to be scanned are extracted from the logical copy using pre-existing knowledge of the VM operating system similar to forensic techniques employed by Volatility [24]. Segment data meta-data is sent to the scheduler and instructs the extractor to send/store only unique page data. This prevents shipping large amounts of data over the network.

After the necessary page data is shipped, the scan scheduler beings pre-processing by reconstructing stored page data into memory segments for each VM. Each segment can be naively scanned with traditional virus scanners however the total size to scan ($P_{size}$) can be very large.

To address scanning large amounts of data, file based virus scanners use hashing to identify previously scanned data. Files hashed to known values do not need to be re-scanned. This technique is most beneficial on static data. Unfortunately, memory segments are often unique due to its dynamic nature making this simple optimization less effective.

We observe that while many memory segments are unique, many segments are similar to some other segments within a VM. For example, the Windows library ieframe.dll can be found mapped across many processes in a running system (e.g., Internet and Windows Explorer). Memory segments for ieframe.dll will share identical PE headers and code sections but will differ in their data sections. As a result, a naive scanner will simply identify both segments as unique.

Using simple hashing techniques to eliminate scanning duplicate data is not effective for memory. Furthermore, false positives can arise from scanning non-present or zeroed data due to coarse grained virus definitions. SEER employs four optimizations to address the above issues:

1. Only globally unique page data is sent off scanned VM hosts. This obviates shipping all snapshot memory and reduces the processing time to extract memory hence reducing impact to VMs, hosts, and network.
2. SEER matches identical prefixes across all memory segments. Matched prefix data determines where duplicate scanner computation occurs. SEER uses matched prefix data to scan identical prefixes only once. For example, two libraries extracted from memory will have duplicate header data but contain different runtime data. SEER will match the header as a prefix and only scan the header data once instead of multiple times.
3. Memory segments are normalized intra-VM to increase similarities between segments. Normalizing memory segments increases the lengths of matched prefixes. This results in less data being scanned.
4. The SEER scanner is provided meta-data and thus can avoid unnecessary computations by not processing non-present data.

The above optimizations allow SEER to efficiently scan unique data reducing the computation required to analyze the full set of memory segments (C4).

SEER leverages existing virus scanners for identifying and matching known malware signatures in memory. Thus, SEER’s effectiveness at identifying malware is reliant on the adopted virus scanner’s signature database and signature matching techniques. SEER does not look to improve matching of existing scanners but to provide a novel architecture for extending existing techniques to efficiently scan memory.

Assumptions and Threat Model: We assume that an attacker has full control over the guest VM. However, attackers wish to remain stealthy and avoid detection thus avoiding behaviors which raise suspicion such as attacks on availability or resource exhaustion.

3.2 VM Host Memory Segment Extraction

Next we discuss the process of memory extraction and segment data shipping off VM host machines. Memory extraction is built on DACSA to limit the impact to host and guest environments. We add discussion on shipping memory segments not previously discussed in DACSA [7].

3.2.1 DACSA: Fast Snapshots and Extraction

DACSA is a decoupled architecture for virtual machine memory analysis. DACSA introduced a reliable fast memory snapshot technique using Copy-On-Write to create a logical copy of Virtual Machine memory allowing memory to become a data source for analysis. Memory forensic techniques are then applied to extract security relevant information. DACSA showed to have minimal impact on host and guest operations during the snapshot and memory extraction process. Due to space constrains we limit the discussion on fast snapshotting and memory extraction. We encourage the reader to read previous work on DACSA [7].

3.2.2 Carving Segments from the Logical Copy

Memory segments\(^1\) are carved from the logical copy based on the VM guest OS semantics. For Windows, memory segments are obtained from walking each process’s Virtual Address Descriptor (VAD) tree structure [5]. The VAD tree is a binary tree where each node represents an allocated memory space by the process and thus a single memory segment. Each VAD tree node contains a starting and ending virtual address specific to the process. We extract memory segment data using these addresses and the process’s page-tables.

3.2.3 Shipping Memory Segment Data

Each segment is processed during segment extraction page-by-page using the VAD starting and end address. A hash of each page is generated to determine if the page data needs to be sent to the schedule. First, the hash is looked up locally. If not found, the hash is added to a batch of hashes to send to the scheduler. Once the batch is sent, the schedule checks each page in a global hashtable and returns which pages are

\(^1\)We use “memory segment” to refer to the carved unit of memory, which does not correspond to an x86 segment.
Figure 2: Two binaries that differ at byte $X$ have an identical $X - 1$ bytes prefix. Binaries are split into windows size $s$. Hashes of window $w_{n-1}$ identifies the difference.

needed for scanning. If the page hash value has been stored before, meta-data for the hash is stored with the hash identifying the VM, segment, and offset. The scheduler then uses the hash to reconstruct the segment for pre-processing and scanning. This process is similar to VM memory deduplication techniques [8].

3.3 Segment Similarity Detection Off-Host

For segments that are similar but not identical, we observe the scanner may unnecessarily duplicate scanning effort. For example, Figure 2 depicts two identical binaries that contain only a single byte difference at index $X$. A traditional naive scanner will identify each segment as unique and scan each segment separately from low bytes to high. We observe that after each scan instance has processed the first $(X-1)$ bytes in the example, the scanner signature matching states will be equal as the prefix $(X-1)$ bytes are identical for both binaries. Thus, the scanner will have duplicated matching computation for this prefix.

Prefix information provided to the scanner allows a single unique prefix to be scanned only once across similar segments. This is beneficial for binaries used across many processes and machines as most will be unique but share some identical header and code sections.

3.3.1 Prefix Matching

SEER needs to quickly identify common prefixes across all segments. While byte-by-byte comparison provides very fine-grained matching, it comes at a large cost. We are only concerned with identifying identical prefixes which are aligned to the beginning of memory segments.

To quickly identify differences in data, previous techniques such as fuzzy hashing [11] and rolling hashes [22] have been proposed. Unfortunately, these techniques do not lend themselves to finding the longest prefix in pseudo-random data. Specifically, the effectiveness of these approaches relies on finding a suitable trigger. Instead, we choose a more straightforward approach which uses a rolling window to approximate the longest prefix.

Figure 2 provides a reference for partial segment hashes, which we discuss next. After segment extraction, each segment is divided into $n + 1$ non-overlapping parts of data using meta-data to reconstruct segments. We refer to this data as window $w$. The $i$-th window, $w_i$, represents bytes from $(i \times s)$ to $((i + 1) \times s) - 1$ where $s$ is the window size. Each $w_i$ is hashed by the scheduler in order from $i = 0$ to $i = n$. Window hashes, window offsets $(i \times s)$ and size (to handle boundary cases) can then be compared across segments beginning from low to high index. If two hashes at an index do not match, the index identifies the longest prefix for a pair.

Encoding Prefix Information: SEER needs to quickly determine the longest prefixes across all segments and store relevant information for scanning. We achieve this through a combination of hash tables and building a rooted directed tree (e.g., as in Figure 3). The rooted directed tree naturally represents prefixes by encoding information in each node. As the tree is walked from the root towards children, data for segments can be processed and read in linear fashion.

Prefix information is encoded in the tree nodes. Each node represents a unique prefix and maintains several pieces of information used for scanning and segment association. Specifically, nodes contain (1) associated segment ids (SUIDs) which track matched segments, (2) a range to explicitly define where the matched prefix starts and ends for the associated SUIDs, and (3) a hash table for looking up child nodes. To track hashes and data for each node, window hashes are stored for each index represented by the node. These hashes are used to later build and retrieve the data stored from segment extraction.

Algorithm 1 depicts how the tree is built using memory segments and windows. As windows are processed by the scheduler, the tree is walked and updated. For processing new segments, the walking begins at the root node (line 1). If the received hash’s index does not fall in the current node’s explicit range (line 4) then the hash value is not represented by the current node as a previous prefix has been defined for the hash’s index. As a result, the current node’s child hash table is searched for an association (line 5). A new node is added if no child node with the hash and corresponding index is found (line 7). If the hash matches a child, the hash is associated with the matched child node (line 6). During association, the SUID is added to the node’s list of SUIDs with common prefix and matched indexes. If the hash’s index falls in the current node’s explicit range (line 8), the node’s stored hash for index is compared (line 9). If identical, the associated SUID is updated for the node (line 10). Otherwise, a new prefix is identified and the current node is split at the window index creating two child nodes (line 11). To quickly add new unique prefixes, each node stores the first SUID used to create the node. If the SUID for the window hash is the same as the node’s first SUID (line 2), the node only needs to update the current node with the window hash and index information (line 3). Such
is the case when the root node contains no children.

Figure 3 depicts a prefix matched tree with five unique memory segments. Segments with SUID 1 and 4 represent a mapped binary. We see that the longest prefix for 1 and 4 is found at offset 0x1000. Thus, the scanner would only need to scan the first 0x1000 bytes of SUID 1 and 4 one time instead of twice. SUID 2 represents a unique segment as reflected by a node with no children. SUID 3 and 5 represent a data file for browser history. We see that the longest prefix is found at offset 0x7000 when SUID 3 ends. SUID 5 is added as a child of SUID 3 as SUID 5 has an extra 0x1000 bytes in the segment. In this case, the scanner would only scan the first 0x7000 bytes of SUID 3 once. The total savings dictated by the tree is 0x8000 bytes.

3.3.2 Binary Normalization

Binarics mapped across processes often provide opportunities to match large prefixes, as per our 1iframe.dll example. This property allows additional opportunities to deduplicate scanning work. We found that binaries have a semi-consistent structure in which static headers and code resides at the beginning. While nothing prevents code sections from being located at the end of binaries, this pattern is default for common compilers. This attribute of binaries ensures prefixes match at least header and code sections for segments with the same backing file on disk.

Unfortunately, we found that beginning memory is not always identical due to how the underlying OS manages memory for mapped files. For example, a page may be marked not present in a process’s page table but mapped in another. As a result, we use a technique for normalizing binaries to extend the length of matched prefixes.

Filling Non-Present Pages: OS memory management units employ on-demand paging to reduce the memory footprint of processes and improve performance. Only processes accessing the page has its pagetable updated to include the new physical page mapping. As a result, pagetables between processes are not synchronized for common shared binaries.

The lack of inter-process pagetable synchronization impacts prefix matching for memory segments. For example, consider two processes that both use the 1iframe.dll. The third page of 1iframe.dll is accessed by process P1 but not P2. As a result, the third page for P1 will be marked present while the same page for P2 is non-present and the corresponding window hashes (w2) will not match. The longest prefix for P1 and P2 will be the index of w2.

SEER conservatively normalizes memory segments by filling in non-present pages with pages found present in other segments that are mapped to the same binary file. During memory extraction, segments are grouped by binary headers, corresponding load addresses, and size. In Windows, binary files loaded from the same file on disk will all be placed at the same address in memory by the default loader.

Once grouped, each segment’s data is walked to identify present pages and indexes. If a page is present, the status is checked to see if a previous data page has been stored for that index. If previously stored, a hash of the data is compared to ensure equality. In some cases, present data at a page index may not be equal and conflict across segments in the group. This occurs if a page has been modified by a process, for example, when modifying data or malicious code manipulation. In this case, we conservatively mark the page status for the page index as unknown and do not use the page to fill non-present pages for groups of binaries. Page filling occurs with regards to given VM.

3.4 Prefix Matched Scanning

After all memory segments have been processed, the scheduler contains a tree of nodes representing unique prefixes covering all segments. Each node in the tree contains information used by the scanner to scan the unique prefix.

To scan all segments, each node must be processed and data represented by the node scanned. We process each node by walking the tree in a Depth First Search manner. As the scanner requests data to scan, node attributes are used to determine which data to provide. A node boundary is reached when the scanner asks for data at an offset greater than the current node’s end address. As a result, the scanner takes actions to walk each child.

Each child node to be processed requires the same scan state as the parent. Thus, the scan state must be saved or duplicated so each child can have its own copy to continue scanning correctly. To facilitate coping scan state, the scanner forks execution for each child node. The last child does not fork, but only assigns the last child node for processing.

Ignoring Non-Present Memory: Windows of segments found to be non-present can be skipped by the scanner as they provide no information for matching. This provides two benefits: (1) it increases scanning efficiency and (2) it reduces the possibility of false positives occurring that match for non-present regions. When a non-present window is to be scanned, the scanner simply updates the index to be read to the next present page thus preserving scanner context.

Scanner Specific Considerations: Scanning with prefix matching assumes that memory is scanned in a linear fashion. This ensures all matching efforts for the previous prefix has been complete prior to forking. Thus the scanner should not peek at data ahead beyond the current window. Reading data in non-linear fashion will result in walking child nodes prior to matching efforts thus preventing savings from scanner effort deduplication.

3.5 Implementation

We prototype SEER using QEMU [1] v0.13.0 with KVM.
virtualization software as the host. Our host is Ubuntu 12.04
64-bit and VM environments are Windows 7 SP1 64-bit.

The memory extractor and scheduler are configured to
allow both storing complete memory segments to network
storage or sending only unique data windows as discussed in
Section 3.2.3. For our evaluation, we save complete memory
segments to allow for better support for evaluating different
configurations.

SEER requires minor modifications to virus scanner soft-
ware to enable best efficiency and accuracy. Thus, we adapted
ClamAV, a popular open source virus scanner, to the SEER
architecture with minimal modifications by hooking I/O re-
quests through a shared library. For example, read requests
for scanned memory segments are intercepted and return
specific data based on the current node. If a read request
crosses a scan tree node boundary, the process can immedi-
ately fork and return the appropriate values for each read.
ClamAV virus definitions for PE/COFF binaries support
context aware definitions. For example, virus definitions can
target specific sections for binaries. The sections found in bi-
nary disk files reside at different locations in memory as sec-
tions are often page aligned while sections on disk are block
aligned. Thus we modify how ClamAV interprets PE/COFF
files so that sections are aligned correctly with memory. In
total, we only modified 9 lines of ClamAV code to enable
memory virus scanning with SEER.

4. EMPIRICAL EVALUATION

We evaluated SEER on an IBM System X server with a
Xeon E5540 Quad-Core CPU with 32GB RAM. The server
hosts Window 7 SP1 64-bit VMs configured with 1GB RAM
and 1 VCPU with QEMU/KVM as the host. An IBM
System X server with an AMD Quad-Core Opteron 2384
and 32GB RAM was used for scheduling scans, storing,
and scanning data. A third IBM System X server with an
AMD Quad-Core Opteron 2384 and 16GB RAM was used
to benchmark the host with VMs. These machines are in-
terconnected via two 10Gbps switches, one managing public
network and one for management network.

We investigated the impact of SEER’s components across
the scanning process. First, we performed benchmarks on
host and guest VMs to investigate the impact of fast snap-
shooting and shipping memory. Next, we investigated at
SEER’s impact on matching efforts and total scan times un-
der varying conditions, configurations, and caching. Finally,
we evaluated SEER’s ability to find malware in memory and
discuss false positives.

4.1 SEER Impact on Guest VMs

The guest VM is momentarily paused to create the logical
copy for carving and shipping. We measured the average
time a VM is paused at 0.2112 seconds with a standard
deviation of 0.07359 seconds.

We measured the impact of VM snapshooting and data
shipping (i.e., phase 1) for VMs being scanned and co-located
VMs that are not scanned. We used SpecWeb*2009 [21] to
simulate a webserver under load to measure the reduction in
throughput (MB/s) and response time for a single guest VM
(i.e. VM under test). The VM under test was configured
with IIS, ASP.Net, and the SpecWeb banking application.
SpecWeb was configured to simulate 200 simultaneous users
making at least one request every second. The benchmark
was run under three conditions: 1) no VMs on the host in
phase 1, 2) the VM under test in phase 1, 3) the VM under
test is co-located with VM in phase 1. Table 1 contains the
results. A VM can expect an 8.02% reduction in throughput
and a 9.14% reduction in response time during the 4.9 second
phase 1. A co-located VM not scanned can expect a 0.7%
reduction in throughput and 0.76% in response time during
phase 1. We note here that the average amount of data
was shipped was 153.4 MB and 140.4 MB for these configura-
tions however this number is highly dependent on the processes
and services running. The average time to snapshot and ship
the data was less than 5 seconds under load. The majority
of time was spent shipping data.

4.2 Scanned Data and Scan Times

We investigated SEER under the following configurations:
Naive, Prefix Matched with Normalization (PMN), and
Prefix Matched Normalization ignoring Non-Present data
( PMNP ). Naive refers to an unmodified ClamAV using
full segment hashes to identify previously scanned segments.
PMN provides the potential savings of adopting our tech-
iques for host based memory scanners. PMNP shows the
most efficient configuration for scanning memory that can
be both present and non-present data.

PMN and PMNP configurations were tested with differ-
ent window sizes to investigate window size impact on effi-
ciency. For the smallest window, we used 4KB. This natu-
urally maps to present/non-present page data. We also used
different multiples of 4KB aligned windows. The Naive con-
figuration does not have a window size but hashes the entire
memory segment to identify duplicates.

We configure scanning to be performed serially on a single
processor under all above configurations. While prefix based
scanning is naturally parallelizable, we use a single processor
to avoid unfairly biasing in favor of SEER.

To investigate potential savings under different working
conditions, we scanned VMs under three different operat-
ing environments: (1) idle machine after booting to login,
(2) a loaded webserver running IIS and MySQL with 100
simulated simultaneous clients navigating a PHPBB web
application, and (3) VMs running a set of the following
unique GUI applications: IDA Pro, Chrome, Internet Ex-
plorer 64-bit, Notepad, Windows Update, IIS Administra-
tor, WampServer Installer, OllyDBG, Process Monitor, and
Visual Studio Express 2010. All workloads are run on the
same VM image. Workload (2) looks to investigate the po-
tential impact of a high number of reads and writes but
similar number of processes as workload (1) without loading
into the Windows desktop. Workload (3) pertains to loading
the desktop environment to investigate the impact of unique
processes on the amount to scan and scan times. To remove
non-deterministic nature of memory, carving and scanning
were done on static snapshots of memory for each operating

<table>
<thead>
<tr>
<th>VM Impact</th>
<th>Throughput</th>
<th>Response Time</th>
<th>Snapshot Time</th>
<th>Data</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scanned</td>
<td>8.02%</td>
<td>9.14%</td>
<td>4.90 sec</td>
<td>153.4 MB</td>
<td>8.02%</td>
</tr>
<tr>
<td>Non-Scanned</td>
<td>0.70%</td>
<td>0.76%</td>
<td>4.48 sec</td>
<td>140.4 MB</td>
<td>0.76%</td>
</tr>
</tbody>
</table>
Table 2: Average cost of snapshotting and shipping VM memory for remote scan.

<table>
<thead>
<tr>
<th>Type of VM</th>
<th>1 VM Cold</th>
<th>Time</th>
<th>Shipped</th>
<th>10 VMs Cold</th>
<th>Time</th>
<th>Shipped</th>
</tr>
</thead>
<tbody>
<tr>
<td>Login</td>
<td>6.62 Sec</td>
<td>223.8 MB</td>
<td></td>
<td>5.89 Sec</td>
<td>140.0 MB</td>
<td></td>
</tr>
<tr>
<td>Webserver</td>
<td>7.26 Sec</td>
<td>325.1 MB</td>
<td></td>
<td>6.42 Sec</td>
<td>155.4 MB</td>
<td></td>
</tr>
<tr>
<td>Profiled</td>
<td>8.73 Sec</td>
<td>382.0 MB</td>
<td></td>
<td>7.79 Sec</td>
<td>239.9 MB</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4: Percent Data savings and to scan for a single VM. X-axis is window size.

Data Saved/Scanned: Figure 4 shows the amount of data scanned and saved for a single VM instance across workloads. The x-axis shows the windows sizes in bytes. We see that naive hashing can save 12-14% of data from being scanned across all workloads. For PMN, we see the window sizes have little effect on the amount of data scanned due to the window size saving of 40-45%. PMNP shows the greatest savings of 80-90%. Window sizes have a small impact on PMNP at 5% impact as a result of the smaller windows being able to identify more non-present regions. The differences between Figures 4(a) and 4(b) show that a busier machine does not impact Prefix Matching performance. In fact, we saw that a busy machine had an additional savings of ≈5%. For Figure 4(c), we see that more data needs to be scanned as more libraries and processes are loaded and run. This will affect the total time to scan a VM.

Total Scanning Time: SEER adds processing overhead which needs to be accounted when evaluating the performance of scanning. The overhead of SEER can be found from building the scan tree and forking the scanner thousands of times. To investigate the impact of this overhead, we scan the memory segments for all of the above configurations to investigate the wall time savings of using SEER.

For the idle machine at login and the loaded webserver, the graph shows additional savings by adding VMs to scan at ≈20% and ≈60% for Naive and PMN. PMNP has little additional savings by adding VMs to scan.

For the unique GUI based tests, we see no additional savings across VMs. This is a result of different libraries and processes being loaded for different applications. This indicates that adding VMs to scan only provides benefit when the VMs being scanned are running similar applications.

Figure 7 depicts the per VM total scan times across different configurations. Each graph breaks down the costs for pre-processing (i.e., prefix matching and storage), forking scanner processes, and scanning. Across graphs we can see that the window size impacts total scan times by increasing pre-processing overhead. For PMN configurations, a 4KB window adds 1.5-2 minutes of scan time compared to a window size of 32KB. For PMNP, a 4KB window still adds additional overhead, but this is mitigated by savings during scanning. Overall, PMNP reduces the total scan time by 62-72% compared to Naive for different workloads. Figure 7(c) has a larger total scan time due to the difference in the amount of data scanned.
Figure 5: Percent Data to scan as VMs are scanned together. Windows size fixed to 4KB.

Figure 6: Total scan time across multiple VMs. Window sized fixed at 32KB.

Figure 8: Percent data savings and to scan for warm cache Boot to login VM. X-axis is window size.

Caching: After a VM has been scanned, hashes and status information of the scanned memory segments can be cached to prevent re-scanning identical segments in the future. We looked at the effectiveness caching segment hashes for future scans. Figure 8 and Figure 9 depict the benefits of warm caching for the “Boot to Login” configuration. Figure 8 shows a reduced percent data savings between Naive and PMN with a difference of 5-10% while PMNP and Naive have a savings difference of about 25-35%. Figure 9 shows the scan times for PMN and Naive were equal due to the overhead of pre-processing and forking. However, we still see a ≈50% reduction in scan times for PMNP at a window size of 32KB. We found similar results for both the loaded IIS server and GUI application configurations.

Write Working Set: During the Pre-Process phase in Figures 7 and 9, the QEMU process copy remains resident in memory and the guest VM continues to run. During this time, Copy-On-Write in the host is managing the Write Working Set, set of memory pages which have changed since forking. If the Write Working Set becomes too large, host memory can become depleted due to the dynamic nature of memory. We observed that our Write Working Set changes at rate of 100-300 MB per minute for the first minute. Given, the snapshot and shipping time is typically 4-10 seconds, the Write Working Set is not a concern.

4.3 Malware Identification/False Positives

SEER’s effectiveness at identifying malware in memory is dependent on the adapted virus scanner’s signatures. However, we still wish to verify the SEER correctly extracts memory segments that map to existing virus signatures for memory only malware and file based malware and investigate false positives using existing signatures.

To investigate ClamAV’s ability to extract segments to find malware using existing signatures, we compromised one VM with the file based malware Cerberus RAT and another VM with the Memory-Only Malware Meterpreter. Both malware were configured to infect explorer.exe. SEER successfully found both malware running in the memory of the
Table 3: Malware samples from malc0de.com by type identified by SEER.

<table>
<thead>
<tr>
<th>Malware Types</th>
<th># Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trojan.Adload</td>
<td>2</td>
</tr>
<tr>
<td>Trojan.NSIS.Agent</td>
<td>7</td>
</tr>
<tr>
<td>NSIS.Clicker.Agent</td>
<td>1</td>
</tr>
<tr>
<td>W32.AdInstall</td>
<td>1</td>
</tr>
<tr>
<td>Trojan.IRC.Zapchast</td>
<td>1</td>
</tr>
<tr>
<td>WIN.Trojan.DarkKomet</td>
<td>1</td>
</tr>
<tr>
<td>Trojan.Adload</td>
<td>1</td>
</tr>
<tr>
<td>Adware.Cpush</td>
<td>1</td>
</tr>
<tr>
<td>Trojan.Spy</td>
<td>2</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>17</strong></td>
</tr>
</tbody>
</table>

Table 3: Malware samples from malc0de.com by type identified by SEER.

explorer.exe process showing that SEER can be used to identify both file-based malware and Memory-Only-Malware. We then collected 652 samples of malware from malc0de.com [13]. The samples were scanned with an unmodified ClamAV and 60 samples were identified as Malware. 17 samples were successfully run in a guest VM. SEER was able to correctly identify all 17 samples of malware from memory. Table 3 shows the malware types scanned identified by ClamAV.

To investigate false positives, we recorded all segments identified as malware by our modified ClamAV across each benign scan configuration. This set represents the false positives. We found all observed false positives can be categorized into two types. First, ClamAV marked host based virus scanner definitions as malicious. Specifically, ClamAV identified Windows Defender virus definitions loaded into memory as malware. On disk these definitions files are obfuscated thus do not trigger false positives. This problem can be addressed by white listing host scanner processes in memory. The second false positive type occurs from overloading ClamAV signatures which use MD5 hashes of zeroed data as virus definitions. We found 154 out of 2,056,340 MD5 signatures matched only zeroed data. These signatures should not be used for memory scanning. Upon removing these signatures from the description database and whitelisting scanning processes, no false positives were triggered.

5. DISCUSSION

Deploying SEER: SEER shows the best performance using PMNP configuration with a window size at 32 KB. The performance of PMNP is highly dependent on memory segments containing non-present data. If adapting Prefix Matched scanning for a host based memory scanner, non-present information is not encountered and thus performance would be comparable to PMN.

Security and Privacy Concerns: Memory contains the entire running state of a machine and thus contains very sensitive information such as encryption keys, browser data, etc. SEER extracts memory segments such that finding sensitive data may be easier for malicious parties. For example, a malicious party can quickly identify the webserver process which is known to contain stored SSL certificates. Thus, care must be taken when storing memory segments prior to scanning and properly removing benign data after analysis.

Virus scanners themselves have been known to be a target for attack [17]. Since SEER accesses sensitive data across potentially many VMs, care should be taken to ensure compromise of the SEER scanner itself is mitigated and identified. The simplest solution would be to isolate the scanner to a monitored machine and network.

Alternative Operating System Support: Our current implementation of SEER only supports Windows environments. However, SEER can support other operating systems by porting memory carving techniques for guest operating systems. We believe this to be a straightforward porting effort for well-known operating system structures. For example, Linux have structures called virtual memory areas that map directly to memory segments. For unknown operating systems, VMI techniques can be used to identify internal process structures and virtual memory [6].

Limitations: Currently, SEER does not attempt to extract paged-out memory from VM page-files or swap partitions. While it is technically feasible to extract paged-out memory from disk files, the overhead associated with extraction can have a significant negative impact on both host and guest VM operations. As a result, malware can attempt to avoid scanning by hiding in paged-out regions of memory which can cause false negatives. However, we argue that hiding malware in swap is non-trivial as (1) malware must use a deterministic mechanism to force the operating system to page out malicious content, (2) the deterministic mechanism must be in memory thus providing an indicator for hiding, and (3) scanning of memory cannot be bypassed.

SEER relies on memory forensic techniques to find and walk kernel data structures to extract memory segments. These techniques are commonly used by memory forensic tools [24] and popular Virtual Machine Introspection (VMI) solutions [6, 9]. However, walking kernel data structure is potentially vulnerable to Direct Kernel Object Manipulation (DKOM) attacks [3]. Unfortunately, identifying and preventing all DKOM attacks remains an open problem.

SEER provides detection of memory compromise but not protection. Thus, SEER cannot prevent infection of well-known malware or exploits. While security products often protect against infection by hooking file operations, exploitation commonly uses memory only techniques. As a result, security products can be disabled without notification once an exploit has code execution.

Finally, there is concern that attackers can allocate large memory segments by simple virtual memory allocation. SEER mitigates this potential impact by providing page status to the scanner to bypass non-present memory. To force SEER to scan data, the attacker must fill allocated memory with random data thus requiring the scheduler to store unique data. However, this process would deplete system resources exposing malicious activity.

Using Existing Virus Scanner Signatures: We show in our evaluation that existing signatures for file based scanning can be used to identify malware in memory. This obviates creating all new signatures to adapt to memory scanning. However, the effectiveness for matching viruses is dependent on the correctness and precision of signatures. Furthermore, signature based scanning has known limitations and thus SEER suffers from the same limitations.

6. RELATED WORK

Jiang et al. [9] provided virus scanning for individual virtual machines by scanning files on running Virtual Hard Drive (VHD) and physical memory from the hypervisor. Unfortunately, scanning from the hypervisor can impact host and VM performance. Also, scanner vulnerabilities can lead
to complete system compromise putting other VMs on the host at risk [17]. SEER instead performs scanning off host in an isolated environment while scanning virtual memory to correctly match virus definitions.

Wei et al. [25] presented scanning offline logical VHDD stored in a customized deduplicated image library. The authors were able to show a reduction in virus scanning efforts by only scanning unique files across all logical VHDD. Similarly, Soules et al. [20] discussed how to efficiently schedule virus scanning for an enterprise network by scanning unique files only once. These approaches rely on the static nature of files on disk to reduce scanning efforts. However, SEER instead focuses on efficiently scanning dynamic memory for live running virtual machines.

Oberheide et al. presented CloudAV which describes how to increase the effectiveness of file virus scanning by using multiple pieces of scanning software [17]. Services such as VirusTotal, Jotti, and NoVirusThanks provide multi-scanner virus scanning similar to CloudAV without an end client. CloudAV and similar tools rely on data files being similar across machines and thus only unique files need to be scanned. However, the authors do not discuss how to handle dynamic data. SEER instead provides a new technique to increase scanning efficiency of memory.

Trend Micro recently released a VMWare appliance called “Deep Security” capable of agentless virus scanning of virtual machine files using a central virtual appliance for VMWare [4]. Unfortunately, Deep Security only supports file based scanning and relies on the static nature of files to enable scalability. In contrast, SEER tackles the hard problem of enabling virus scanning of highly dynamic data.

7. CONCLUSION

In this work we proposed SEER, an architecture for enabling Memory Virus Scanning-as-a-Service for virtualized environments. SEER overcomes several challenges of memory scanning by transparently and efficiently scanning memory outside of the context of the analyzed VM. SEER quickly and correctly acquires virtual machine memory and we show in the empirical evaluation our technique has minimal impact on VM operations and hosting environments. We propose a new approach for scanning memory by normalizing memory, matching similar prefixes across memory, and then scanning based on prefix data. Our results show that our approach has a 72% reduction in wall time and scanning efforts compared to existing naive scanning approaches. Our approach can be directly applied to existing host based memory scanners to reduce scanning efforts by up to 46%.

8. REFERENCES