Near-Memory & In-Memory Detection of Fileless Malware

Marcus Botacin*
mfbotacin@inf.ufpr.br
Federal University of Paraná
(UFPR-Brazil)

André Grégio*
gregio@inf.ufpr.br
Federal University of Paraná
(UFPR-Brazil)

Marco Zanata Alves*
mazalves@inf.ufpr.br
Federal University of Paraná
(UFPR-Brazil)

ABSTRACT

Fileless malware are recent threats to computer systems that load directly into memory, and whose aim is to prevent anti-viruses (AVs) from successfully matching byte patterns against suspicious files written on disk. Their detection requires that software-based AVs continuously scan memory, which is expensive due to repeated locks and polls. However, research advances introduced near-memory and in-memory processing, which allow memory controllers to trigger basic computations without moving data to the CPU. In this paper, we address AVs performance overhead by moving them to the hardware, i.e., we propose instrumenting processors' memory controller or smart memories (near- and in-memory malware detection, respectively) to accelerate memory scanning procedures. To do so, we present MINI-ME, the Malware Identification based on Near- and In-Memory Evaluation mechanism, a hardware-based AV accelerator that interrupts the program's execution if malicious patterns are discovered in their memory. We prototyped MINI-ME in a simulator and tested it with a set of 21 thousand in-the-wild malware samples, which resulted in multiple signatures matching with less than 1% of performance overhead and rates of 100% detection, and zero false-positives and false-negatives.

CCS CONCEPTS

• **Computer systems organization → Processors and memory architectures**; • **Security and privacy → Intrusion/anomaly detection and malware mitigation**; *Software and application security.*

KEYWORDS

malware, antivirus, processing in memory, pattern matching

ACM Reference Format:
Marcus Botacin, André Grégio, and Marco Zanata Alves. 2020. Near-Memory & In-Memory Detection of Fileless Malware. In *The International Symposium on Memory Systems (MEMSYS 2020), September 28-October 1, 2020, Washington, DC, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3422575.3422775

MEMSYS 2020, September 28-October 1, 2020, Washington, DC, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8899-3/20/09…$15.00
https://doi.org/10.1145/3422575.3422775

1 INTRODUCTION

Damages caused by malicious software range from the exposition of sensitive information to financial losses (e.g., ransomware [72] steals billions from their victims). The most deployed countermeasure against malware is the anti-virus (AV), which inspects files on disk (usually at process/file creation time) to match segments from a list of known malicious byte-sequences (signatures) [75]. To thwart AV detection, cyber-criminals recently started to make use of fileless malware, which infects the image of loaded processes completely from the main memory [19, 74]. Since fileless malware are not written on disk at any moment of their operation, they do not trigger the usual disk-based AV scanning. Moreover, malicious code is injected into already loaded benign processes, making binary scanning at load time ineffective.

To address fileless malware, AVs started to perform memory scanning, i.e., signature searching inside loaded images of processes [37]. While effective, this procedure is memory access-intensive, since AVs need to constantly lock and poll system memory to detect hijacks of benign processes during their execution. Hence, AVs have to handle the inspection rate: more frequent checks may detect signatures on transient states [50], but impose very high performance penalties; sparser checks have less significant performance penalty, but are susceptible to attacks whose signature patterns appear only in the interval between two checks. This scenario creates an urgent need of more efficient memory pattern matching mechanisms that allow for continuous inspection (preventing transient attacks with acceptable performance overhead).

Recent advances on Ultra Large-Scale Integration (ULSI) and mixed logical and DRAM layers inside 3D-stacked chips using Through Silicon Vias (TSVs) [55] led to the concepts of near-memory and in-memory processing: memories gained the ability to perform basic in-place computations without moving data from RAM to the main CPU, leaving the main processor free for more complex tasks. This created an opportunity for making more efficient AVs by taking advantage of near-memory and in-memory capabilities.

In this paper, we propose to instrument memory controllers of current DDR-powered CPUs or inside smart memories (e.g., Hybrid Memory Cubes - HMCs, High Bandwidth Memories - HBMs [42]) to create a novel hardware-based malware signature matching mechanism able to detect fileless (and traditional) malware without moving data from RAM to the CPU. To implement a lightweight checking procedure, we relied on the unexplored time-window between memory buffers write and read requests for the same addresses in both DDR and smart-memory memory controllers. Thus, we can perform almost inexpensive pattern matching routines, ensuring an invariant in which each piece of read data had been previously scanned at the time it was written. As far as we know, we are the first researchers to propose a hardware-assisted malware signature matching procedure using either near-memory or in-memory processing techniques.

Our main contributions are: **(i)** we propose MINI-ME (Malware Identification based on Near- and In-Memory Evaluation), a novel hardware-assisted malware detector implemented inside the memory controller; **(ii)** we observed inside the memory controller a very frequent time window of write-to-read operations to the same address, which we used to effectively detect malware in memory, as well as to reduce software-based AVs overhead; **(iii)** we detect cache-resident malware by reinforcing a write-through policy on memory pages affected by Self-Modifying Code (SMC). This policy imposes negligible overhead for legitimate SMC code that modifies non-cached pages (e.g., Java and Python); **(iv)** we simulate MINI-ME's operation and explore its design space to identify the best signature sizes regarding detection rates and performance.

MINI-ME accelerates in-memory pattern matching and detects malware with an additional bit to the page table, causing detection notifications to be handled via standard page-fault routines. We obtained zero false-positives (FP) with a deterministic matching procedure, and an FP rate smaller than 1% with a probabilistic matching procedure based on Bloom filters, also reducing the storage required for deterministic matching. Both procedures had overheads smaller than 1%, showing MINI-ME's feasibility for actual scenarios.

This paper is organized as follows: in Section 2, we motivate our work; in Section 3, we present background concepts that substantiate our developments; in Section 4, we introduce the design of MINI-ME; in Section 5, we present MINI-ME's implementation details; in Section 6, we show MINI-ME's evaluation through multiple criteria; in Section 7, we discuss MINI-ME's contributions and the future of in-memory threats; in Section 8, we discuss related work and how they differ from MINI-ME; finally, we draw our conclusions in Section 9.

2 MOTIVATION

In this section, we present experiments to demonstrate the performance bottlenecks that we aim to mitigate and our reasoning about the adoption of a hardware-assisted solution for it.

Statement 1. Software-based, continuous memory scanners impose overhead regardless of their implementation. AVs can implement memory checking procedures using three distinct approaches: (i) dumping the running processes' virtual memory; (ii) dumping full userland virtual memory; or (iii) dumping full system physical memory.

To *dump running processes' memory*, userland AVs first enumerate all running processes (`EnumProcess` API [43]), then open handlers for the targeted processes (`OpenProcess` API [44]), and finally read their memory contents (`ReadProcessMemory` API [45]). Due to the need for calling multiple functions and retrieving tokens for processes inspection, this approach imposes a significant system slowdown. In addition, as processes are handled through their virtual memories, the overhead caused by the explicit OS boundary checks [48] and userland-kernel transitions due to OS API calls is unavoidable and the spent time cannot be masked among other operations because of the inspected processes must be suspended (locked) by the inspection procedure to gather information from a consistent state. An advantage of this approach is that the AV can select specific processes and/or processes' memory regions to dump and inspect.

In the second approach, the AV follows the same strategy previously described, but do not filter memory regions or processes, thus *dumping all userland-allocated memory* resident in the RAM.

When *dumping physical memory*, the OS simply asks OS to collect all memory addresses data without any boundary control. Since this approach does not require explicit OS checks or processes enumeration, the dump procedure tends to be faster. As a drawback, as no memory boundaries are provided by the OS, whole system memory is dumped, which may increase the dump file size on systems having large memory capacities. To implement this type of dumping, AVs are required to load a kernel driver, thus making it a less popular approach than virtual memory dump approaches implemented at userland. However, despite implementation issues, this approach can also be implemented by AVs, as it is already employed in forensic tools [25, 57].

To evaluate the impact of the aforementioned approaches on actual systems, we have deployed them into a 4-core Intel i7 Skylake, 2x4 GB 2133 MHz DDR-3 DRAM machine running MS-Windows 7. To evaluate the cost of dumping only the running processes, we used the `ProcDump` tool [24] while sequentially running the benign applications from the SPEC CPU 2006 benchmark suite [69] to populate the memory regions, consuming a total of 800 MB of used memory pages. To evaluate the cost of dumping the full userland memory, we repeated the experiment now completely dumping a process which dynamically allocates a virtual-memory-based buffer of the same size of the total RAM. To evaluate the physical dump approach, we used the frameworks OSForensics [57] and Rekall [25] while also running the SPEC benchmark applications. In all tests, we limited the number of dumping (forensic tools) and populating (SPEC) threads to one to minimize the pressure on the memory controller. Figure 1 shows the average results obtained from 10 independent dumps for the most-affected, the average-affected, and less-affected SPEC applications.

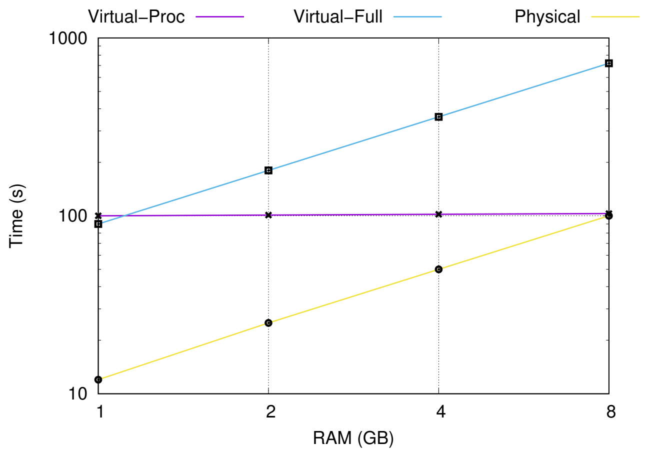

Figure 1: Memory dump time for distinct software-based techniques and memory sizes. They impose non-negligible performance overhead regardless their implementation.

We observe that, as expected, the time spent dumping the full userland virtual memory (`Virtual-Full` curve): (i) grows linearly as the total amount of dumped memory increases; and (ii) is the longest among all approaches, due to the total amount of translations and OS invocations. In turn, limiting the total amount of dumped memory to only the running processes memory (represented by the `Virtual-Proc` curve) results on dump speed up, since the same amount of memory (the processes-allocated pages) is always dumped regardless of the total amount of memory present in the system. For the case where approximately the same amount of memory is dumped in both approaches (800MB vs. 1GB), the full dump approach is a bit faster because dumping contiguous pages results on a higher throughput than enumerating sparse process pages.

The fastest approach, however, for all memory sizes, is the physical memory dump (`Physical` curve). We can observe that dumping physical memory is one order of magnitude faster than dumping virtual memory. The physical dump is faster even when only targeted processes are dumped. As for the virtual dump approach, the physical dump cost also grows linearly as the memory capacity is increased, because the whole-system memory is always dumped. Despite being faster, the dump's cost is non-negligible, therefore, although AV memory scanning capabilities could be more efficient by using physical memory dump approaches, the overhead imposed by performing memory dumps is unavoidable to any software-based approach.

Statement 2. Software-based, continuous memory scanners impose a non-negligible overhead. Regardless of the adopted approach, we can notice that the imposed performance penalty overhead to suspend system's execution and perform a memory dump for AV scanning is non-negligible. Therefore, improving existing software-based AVs require more than improving their implementation (moving from virtual-memory dumps to physical memory dumps) but changing their paradigm (for instance, moving from software to hardware implementations), as any software-based implementation will impact into the system's performance. To further evaluate this impact in practice, we measured the overhead that a whole memory scan performed by a real AV imposes to the execution of third applications running in the system under scan. For such, we measured the individual overhead imposed to each application from the SPEC CPU 2006's benchmark suite (average of 10 executions) when executed along with a continuous memory scan by Clamwin [16], a Windows version for the open-source ClamAV with memory scan and real-time support [15]. We selected ClamAV because it is an open-source AV solution, thus easing the instrumentation required for performance monitoring. The AV was executed with all default configurations (e.g., default signature size, scan intervals, so on). Figure 2 shows the overhead imposed to the benchmark applications which respectively were most (top 3) and less impacted (top 2) by the AV execution.

We identified that ClamWin's memory scans imposed performance overheads from 5% (in the best case) to up to 100% (in the worst case) even on legitimate applications, using a 4-core processor, executing only 2 threads (from the benchmark and the AV). This overhead is unavoidable for software-based AVs because they need to use the memory-CPU buses to retrieve data from the main memory and store them in CPU caches, thus causing a

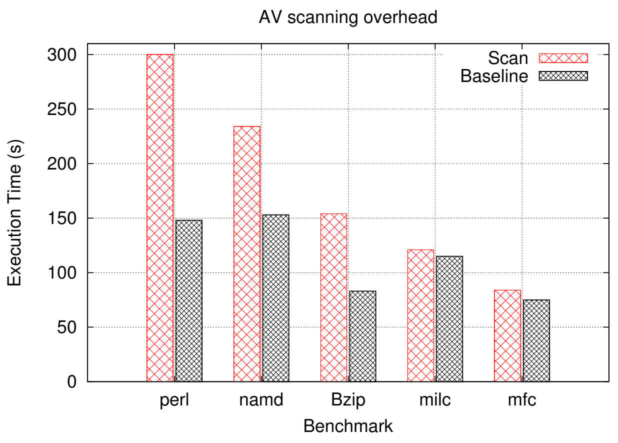

Figure 2: In-memory AV scans worst-case and best-case performance penalties. ClamWin's scans imposes penalties from 5% to up to 100% even on benign application's executions. Any software-based AV will impose such significant overhead as they compete for system resources with all other running system's applications.

resource competition with legitimate applications running in the same system. Therefore, implementing a memory-supported AV would significantly reduce the performance impact caused by any software-based AV.

Statement 3. Existing hardware features provide detection triggers but do not eliminate the performance overhead. We have so-far shown that software-only solutions will always impose a significant performance overhead. We now investigate the application of existing hardware features to support these solutions in mitigating the performance overhead. In particular, we discuss the reliance on the MMU.

Malware detection can be understood as two distinct tasks: (i) identifying an inspection opportunity for a given resource (e.g., process, page, pipe, so on); and (ii) scanning the pointed resource for infection identification. The existing Page Faults (PFs) handler might be a good trigger for the first task, as it can indicate read, write, and execution attempts to individual memory pages. Forensic solutions often operate on a copy-on-write (CoW) manner [41], unsetting MMU flags to intentionally cause a PF to be handled under their control. This type of implementation requires a deep level of kernel access and thus we are not aware of any AV leveraging this technique.

Even if AVs were able to implement a complete CoW mechanism, this would not completely solve the problem, as the second task of the detection process would still be required to be performed by the software components. Forensic procedures have the significant advantage to be allowed to perform offline checks. In turn, AVs are required to perform online detection to block the threats as soon as possible. Therefore, the AV would still be running a significant amount of code during each PF handling.

Table 1 shows the impact in the performance of the same SPEC applications shown in Figure 2 when blocking on PFs by a distinct number of cycles. Although the relative number of PFs is short in

Table 1: Blocking on Page Faults. The performance impact is greater as more complex is the applied detection routine.

Benchmark	Cycles	PF	5K	10K	20K	30K
perf	187G	1,8M	4,74%	9,48%	18,96%	28,44%
mcf	69G	375K	2,72%	5,45%	10,89%	16,34%
milc	556G	1,2M	1,05%	2,10%	4,21%	6,31%
bzip	244G	170K	0,35%	0,69%	1,38%	2,08%
namd	491G	325K	0,33%	0,66%	1,32%	1,98%

comparison to the total number of spent cycles, the overhead might still be significant depending on the deployed detection routines. The imposed overhead is increased when the detection routines are more complex and thus take more cycles to be processed. The overhead can reach 28% for `perl` if we consider a routine that takes 30 thousand cycles, which can be reached for complex regular expressions implemented at high level [11].

Therefore, in this paper, we look for a mechanism that provides both a good trigger and a parallel processing capability to mitigate the performance overhead imposed to all detection steps. We following show how these goals are achieved via the application of instrumented memory controllers.

3 BACKGROUND

In this section, we present background information about the concepts that support our developments. We first introduce the concept of fileless malware, the threats we want to defend against. We second introduce the concept of smart-memories, as they provide the basis for our solution, We finally discuss the write-to-read window, the time frame exploited by our solution to mask the inspection overhead.

Fileless malware. Traditionally, AVs scan file contents for malicious patterns. Although it used to be enough for most scenarios, since even newly created processes were loaded from files downloaded in the disk, the emergence of the so-called fileless malware changed everything. Also known as Advanced Volatile Threats (AVT), this type of threat can infect a running process without having a disk counterpart, thus being undetected by file-based AVs. Fileless malware infections are enabled, for instance, by memory writes from Javascript code [73], which writes process memory with malicious code and perform runtime thread creation. The concept of malicious software that operates solely from the memory is not new (it was proposed in the '80s [17]), but its implementation was flawed: since it does not have a disk correspondent, it is not persistent, which causes the attacker to lose the malware's control in the event of a reboot. However, as modern machines hardly often reboot, such attacks become practical. Many fileless-based attacks have been recently reported [20, 74] and, to handle their threat, AVs must periodically scan system memory, in addition to disk files at process creation time. Additional scanning imposes the overhead of including the verification of legitimate processes that could have been infected through their memory spaces, as observed by Kaspersky [37]. In this work, we propose an efficient way to perform memory checks able to detect in-memory malware without adding the significant overhead imposed by current software-based AVs.

AV signatures. The most widespread detection technique leveraged by AVs is the signature matching [28]. In this approach, the AV looks for byte patterns known to belong to malicious samples. These patterns often correspond to the instruction bytes which implement a given malicious behavior. Code 1 presents an example of a signature generation procedure from a malicious code snippet responsible for implementing a debugger evasion technique based on the internals of the Windows native library `IsDebuggerPresent` [47], which is often found in many malware samples [10]. When compiled, the malicious C code presented in Code 1a will produce the assembly code presented in Code 1b (see details in [10]). When loaded in memory, such code will be represented by the byte sequence presented in Code 1c. Therefore, this byte pattern would be the malicious signature itself. Similar to the AV industry, MINI-ME considers byte sequences as signatures for in-memory malware detection. Over time, signature-based detection had been gradually considered less attractive given a continuous arms-race between attackers and defenders. Malware attackers started to mutate their samples by multiple means, such as using crypters [71], to present distinct signatures than the ones originally identified by AV vendors. Therefore, AVs started to also rely on distinct approaches, such as behavior-based ones [13], more resistant to obfuscation. However, signatures resurfaced recently given the emergence of in-memory, fileless malware. As these can infect even benign applications, behavior-based approaches are not enough for detection as they can be biased by the legitimate host process behavior. Therefore, whereas there are behavior-based approach for fileless malware detection [23], signature matching is currently the most effective way to detect this type of threat with higher accuracy, thus being leveraged by the AV industry [37]. Thus, in this work, we considered signatures to detect in-memory malware samples.

Write-to-read time window. Current memory controllers are composed of multiple queues [34], which allows controllers to implement distinct data handling policies. Each memory controller has at least two distinct request queues (as shown in Figure 3): one for write requests and other for read requests. A typical response time-focused policy implemented by most controllers is to prioritize recent data read requests instead of cache write-back requests. This way, read requests might overlap other read requests for any address. This policy helps the system to sustain higher throughput rates as consuming data (reading memory) is a key computation task for most applications. Despite allowing multiple policies implementation, an invariant must be ensured: memory read commands must not overlap previous memory writes commands for the same memory address, thus keeping context consistency. In other words, it must avoid Read-After-Write (RAW) bypasses. In practice, however, this case is rare and write commands are often not latency-critical, since read requests for the same address often come only after multiple cycles [35, 67]. Therefore, in practice, there is a write-to-read window during memory operations.

Identifying this time window as an AV scanning opportunity is key for our solution since it provides a timing upper bound for a hardware-implemented AV check. In other words, if a scan is triggered along a memory write request command for a given address, an AV can take up to the next, same-address read command completion to perform the scan without causing the memory to delay the response to a further read request. Therefore, by exploring

```
// Windows API
if(IsDebuggerPresent()){
// Attacker Routine
  evade()
```

a: C code.

```
// inline anti-debug asm
mov eax, [fs:0x30]
mov eax, [eax+0x2]
jne     0 <evade>
```

b: Assembly code.

```
64 8b 04 25 30 00 00
67 8b 40 02
75 e1
```

c: Instruction Bytes.

Code 1: Signature Generation for an evasive malware sample.

the write-to-read window, we propose implementing an overhead-free AV scanning mechanism that ensures the invariant that every read data was previously scanned by the time when it was written in the DRAM.

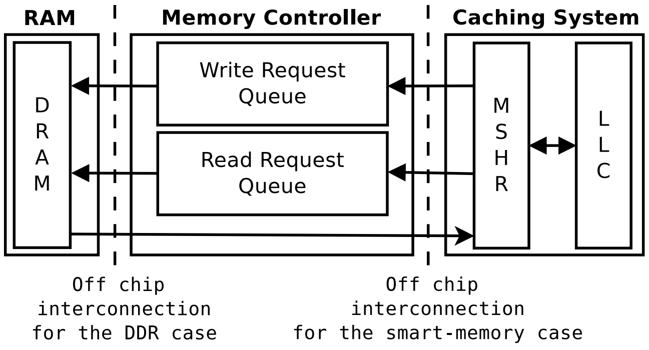
RAM
Memory Controller
Caching System
DRAM
Write Request Queue
Read Request Queue
MSHR
LLC
Off chip interconnection for the DDR case
Off chip interconnection for the smart-memory case

Figure 3: Write-to-Read window. Read requests originated from the MSHR might overlap other memory-buffered read requests for any address, but must not overlap previous memory-buffered write requests for the same address.

4 MINI-ME DESIGN

In this section, we present the design of MINI-ME and present the expected usage scenario.

Threat model: MINI-ME's goal is to perform fileless malware detection in an efficient way, thus enhancing AV scan operations in the case where AVs present significant drawbacks, and not completely replace them on detecting ordinary, disk-based threats, a task which current AVs perform reasonably well. Therefore, we consider that AVs will still implement their own detectors for other threats (e.g., web attacks). Our threat model assumes code injection-based fileless attacks and does not consider code-reuse attacks, such as return-oriented programming (ROP) since these are not handled by AVs but by other system mechanisms [8]. To detect fileless malware, MINI-ME still relies on malware signatures provided by the AV companies, distributed via the Internet as malware definition updates. Due to the signature's nature, the ideal usage scenario for MINI-ME is to counter 1-day attacks, when new threats are recently discovered and no other detection method is available, although MINI-ME can be applied in any scenario. MINI-ME checks malicious patterns both in kernel and userland spaces but privilege escalation prevention and integrity assurance are out of MINI-ME's scope, as authentication and authorization are not AV's responsibility. Although MINI-ME supports any Operating System, we here assumed MINI-ME operation on Windows, as it is the most popular [52] and targeted OS [36] by malware writers. The system that MINI-ME runs on may be supported by HMCs, HBMs, ordinary DDRs or a combination of them.

Architecture: MINI-ME's key concept is to accelerate fileless malware detection by moving AV's pattern matching operation from software to hardware, adding it to the memory controller. MINI-ME does not eliminate the software-AV component but limits it to handle only the malicious patterns detected by the hardware component. On MINI-ME, the memory controller is responsible for automatically and continuously retrieving modified data and comparing it to a database of known malicious signatures. Handling data near and/or inside the memory helps reducing overhead as data does not need to be moved to the main processor to be inspected. When a malicious pattern is identified, MINI-ME invokes a software-based AV component on-demand to decide whether the running process is malicious or not. In case of false positives (FPs), it requests MINI-ME to add such location to a whitelist.

To accelerate AV scans and avoid adding overhead to other application's executions, so important as to perform the matching procedure near and/or inside the memory is to do it in appropriate time opportunities. If we opted to use the ordinary logic layer operations of the smart memories to do so, we could overload it with AV requests and compromise the response time of other CPU requests, meanwhile, we would also require CPU time for the AV trigger such operations. Therefore, we opted to take advantage of the time window normally existent in memory controllers between a write and a read operation (write-to-read window) for the same memory address (see Section 3). The main rationale behind our mechanism is that only modified memory regions need to be scanned. Thus, only data being modified (written) requires a check. Moreover, such detection is only required to be finished whenever the processor reads the written data to execute the malware instructions. Therefore, in most cases, MINI-ME will deliver the scan result along with the read request without imposing any overhead. Overhead is only imposed in rare corner cases, such as for read-after-write requests (see discussion in Section 6).

To implement this model, MINI-ME relies on 3 modules: (1) a userland AV; (2) a kernel driver; and (3) the memory-based AV at hardware. The *userland AV* component is responsible for adding threat intelligence to the system, such as enforcing distinct security policies. Upon starting, it updates its malware signatures definitions from the Internet and load them in the memory controller logic to be matched. When a pattern is matched, the AV is notified and then it decides which action will be taken (process whitelisting or blocking, for instance). By keeping threat intelligence in software, we can still benefit of years of AV industry expertise whereas still improving AV performance by moving the matching to the hardware.

MINI-ME requires a *kernel driver* to allow the userland-hardware communication. The driver is responsible for receiving the userland AV component requests (start monitoring, load signatures and whitelist regions) and forwarding them to the memory controller by writing to memory-mapped memory controller's control registers. The control region is mapped only in the kernel, thus protecting MINI-ME from userland tampering [31], respecting the privileged monitoring principle [62].

MINI-ME's *hardware component*, responsible for checking memory for malicious patterns, is implemented inside the memory controllers and is formed by: (i) the Matching Engine, which can be implemented in several different ways; (ii) the Signature Database inside the Matching Engine to store the malware signatures; (iii) a Malicious bit inside the read packets/commands; (iv) the Malicious Bit Database to identify the malicious memory rows; (v) the Matching Signatures Area (MSA) to store the matched patterns; (vi) the Whitelist Bit Database to identify whitelisted memory rows; and (vii) a Malicious bit for each entry of the Page Table. Database implementation details are described in Section 5.

MINI-ME's Matching Engine (i) incorporates the Signature Database (ii) and queries it for malicious patterns. The database is externally loaded by the driver, allowing signatures updates to occur without hardware redesign, a common drawback of previous hardware-AV solutions (see Section 8).

Usage example: Figure 4 presents MINI-ME architecture and operation (the MSA is omitted for the sake of simplicity). Each time an *income data write packet* is received by the memory controller (**1**), MINI-ME stores the data in the DRAM (**2**) and the Matching Engine in parallel matches the data against the patterns stored in its database (**3**), adding a suspicious flag for the corresponding row in case of detection (**4**). This pattern is also stored in the Matched Signature Area (MSA) if MINI-ME is configured for it. Suspicious memory rows are unflagged after the notification delivery to AV. MINI-ME implements a `Scan-On-Write` (SoW) policy, raising detection notifications only once for each distinct memory write. MINI-ME re-scans a row after a new write, as the memory content may have been modified.

When a *memory read is requested*, MINI-ME reads the data (**5**), adds it into the read packet (**6**), and in parallel it checks if such memory region was scanned and identified as suspicious (**7**) to add a suspicious flag to the read packet (**8**). After assembling the read packet with the suspicious flag, the userland AV needs to be notified. However, as detection occurs on physical memory and AV operates on a process-basis (i.e. virtual memory), there is a semantic gap to be overcome. We bridged such semantic gap by making the suspicious bit available to the MMU. Therefore, each time a page is translated, its suspicious flag is also mapped in the MMU, thus allowing the O.S. to deliver detection notification as an ordinary page fault. As page-fault handlers are aware of virtual memory addresses, these can be mapped to O.S. processes, as is usual in security solutions introspection procedures [8]. This procedure should be repeatedly executed by the userland AV component for every newly created process and after every system reboot to handle the issues related to Address Space Layout Randomization (ASLR) and Position Independent Executables (PIE).

A drawback of relying on the MMU is that it operates over more coarse-grained data (pages) than the DRAM (memory rows). Therefore, multiple matching memory rows are mapped to the same suspicious MMU flag. The detection is further disambiguate by the userland AV, which queries the matching patterns stored in the MSA if required. On the other hand, an advantage of relying on the MMU is that only mapped pages can raise AV detection, thus reducing the overall imposed performance penalty due to detection occurring in non-mapped pages. It also allows distinct policy implementations: As the MMU is aware of page permission bits, the page fault handler may be instrumented to forward detection notifications only for executable pages, for instance. Ignoring data pages not only reduces notification delivery overhead, but also the number of false positives, as code is less diverse than data, thus less prone to random pattern collisions.

After the PF, the detection notification is delivered to the userland AV component, which is responsible to implement a detection policy (see policies in Section 7). The userland component might query the detected patterns in the Matched Signatures Area (MSA) (see memory commands in Section 5) and input them to AV's complex state machines responsible for modeling infections and identifying the malware samples. The userland AV component might then decide, for instance: (i) to immediately block the process execution; (ii) to allow execution to resume and wait for more detection notifications for the same pages to increase its confidence on the detection correctness; or even (iii) whitelist the process execution in case of a confirmed False Positive.

During monitoring, false positives (FPs) may occur due to multiple reasons: spurious data coincidence, a bad signature choice by the analyst, etc (a discussion on signature generations policies is presented in Section 7). If a FP occurs and the memory value remains unchanged, consecutive memory reads would lead to a constant FP detection at such location, which triggers unnecessary AV calls. To prevent that, a whitelisting mechanism should be deployed, so AVs can mark such memory regions as clean after identifying its detection as a FP. If the userland AV identifies that a given notification is a False Positive (FP), it raises a whitelist command for the reporting memory region (**9**) by writing to a specific MINI-ME control region (`0xADDR`) the address to be whitelisted (Notice that in this case, the address to be whitelisted comes from the `DATA` path since the `ADDRESS` path is set to the whitelisting control region). MINI-ME sets the whitelist bit in the Whitelist Bit Database for the address corresponding to the misdetected region. This bit will cause the malware detection check (**8**) to be false, thus not triggering detection notifications for further read requests. The whitelist bit is automatically set off after a new memory write in the same memory address (**10**). Whitelisting a region requires bridging the semantic gap in the opposite direction than the notification (processes to DRAM). For such, the following procedure was designed: Whitelist requests originate as ordinary read/write commands and thus the pointed address is translated to a row address by the memory controller. MINI-ME then traps this request and forwards it to the whitelist database.

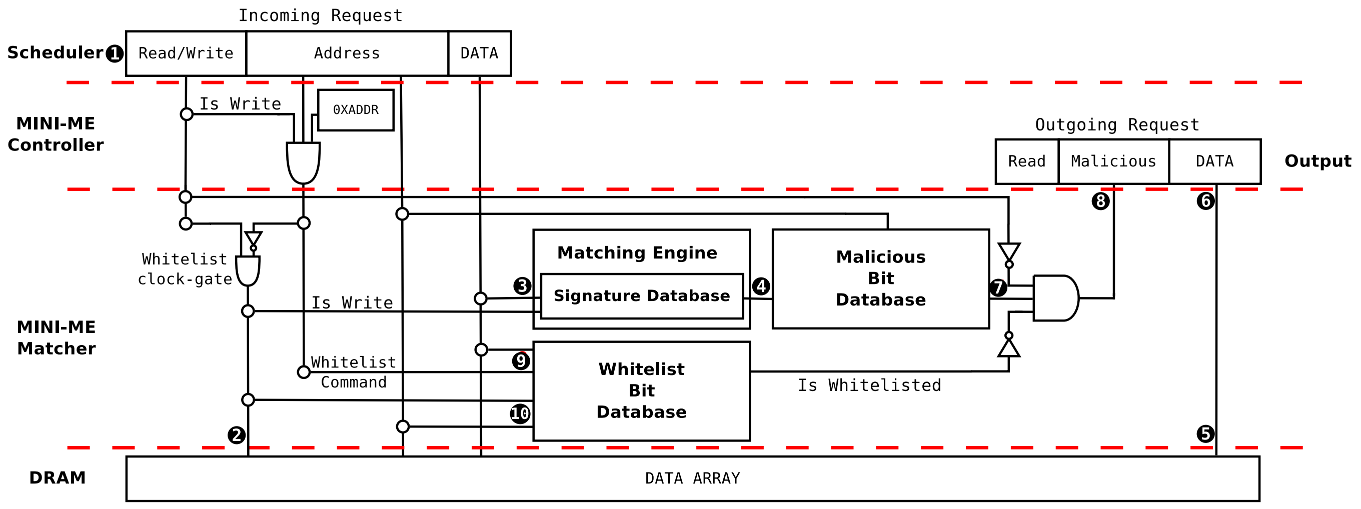

Figure 4: MINI-ME Architecture. MINI-ME is implemented within the memory controller.

5 MINI-ME IMPLEMENTATION

In this section, we present the project decisions for MINI-ME's proof-of-concept. We focus our description on MINI-ME's architectural components, since its software components were implemented as extensively described in the literature (e.g., driver development [46]). MINI-ME implementation used a simulator based on Intel Pin [40].

5.1 Memory-OS integration

The AV and the O.S. should be able to communicate with MINI-ME's instrumented memory logic layer to enable/disable the monitoring mechanism, load signatures and other management tasks. MINI-ME receives commands using mapped memory regions in the same manner the OS use to communicate with I/O devices [68]. Notice that by using memory region mapping we avoid modifying the ISA from the host processor, making our approach fully compatible with existing ISAs (although porting MINI-ME to work with new ISAs is also possible). Once the OS/AV has sent commands to MINI-ME's control memory region, they will be decoded by MINI-ME's intelligence at logic layer. Table 2 describes MINI-ME's control commands. Each command (column I) takes an argument (column II) as immediate to implement a given behavior (column III).

Table 2: Proposed commands allows controlling MINI-ME's detection in a fine-grained manner.

Command	Argument	Behavior
`control`	ON/OFF	Start stop matching
`load`	ADDR	Load Signatures pointed by ADDR
`matches`	ADDR	Check matches in the region pointed by ADDR
`allow`	ADDR	Whitelist region pointed by ADDR

The `control` command is responsible for enabling and/or disabling MINI-ME matching. A request to start matching is only valid after a `load` command to set the signature database. The `load` command copies the bytes pointed by ADDR directly to the internal database. After a match, on can query the memory via the `matches` command to check the matching patterns. In cases where a FP occurs, the region can be whitelisted by setting an `allow` command having the conflicting address as ADDR argument.

Whereas the aforementioned commands allow OS-memory communication, MINI-ME also needs a way to notifying O.S. about suspicious patterns detection. Although smart memories already present a native precise exception mechanism, we opted to not create a new system interruption point but to let the OS to query memory status during an existing interruption, thus reducing the required modifications to the native system architectures. More precisely, we propose making the suspicious bit/flag available to the page table via the delivered outgoing packets. Therefore, whenever a page-fault occurs, the memory provides the requested page and populates the table with the detection flags, thus allowing malware detection to be handled within existing OS page-fault (PF) handlers. Code 1 exemplifies the proposed modification of the PF handler to get suspicious executions notifications.

Listing 1: Modified PF handler. Malicious bit is set when suspicious pages are mapped.

```
void __do_page_fault(...) {
    // Original Code
    if (X86_PF_WRITE) ...
    if (X86_PF_INSTR) ...
    // Added Code
    if (X86_MALICIOUS) ...
```

As the Page Fault handling routines have access to MMU flags, the OS PF handler might implement multiple policies as defined by the userland AV, such as notifying the userland AV about a suspicious page request only when given MMU flags are set (e.g., executable pages only). Moreover, as the Page Fault handler operates in the virtual memory space, it can provide the suspicious memory region address to the userland AV, which allows the AV to identify to which process such region belongs and apply per-processes detection policies.

5.2 Handling self modifying code

Self Modifying Code (SMC) are pieces of code able to mutate themselves at runtime via writes to the instruction memory. As read requests to written data are usually forwarded by the CPU's Last-Level Cache (LLC) Miss Status Handler Registers (MSHR) and not directly delivered to the main memory (see Section 3 for details on data-forwarding), an SMC code could remain undetected in the instruction cache, thus evading MINI-ME detection, if the execution permission flag were not considered. To overcome this challenge, MINI-ME relies on the fact that modern processors require system's MMU to handle writes to executable pages by flushing the SMC payload from the cache and reloading it from the main memory [33], which allows MINI-ME to inspect them. By relying on this characteristic, MINI-ME imposes no overhead to non-SMC code and an almost negligible overhead to benign SMC code, since their pages are scanned **only** when loaded for the **first** time, being considered as "clean" after the first check.

An almost negligible overhead is also imposed to applications that rely on runtime code generation, such as `Java` and/or `Python`, since their JIT engines generate code first by writing to data pages and further turn these pages executable by setting the executable bit for the written page in the MMU, a sufficient time window for MINI-ME inspection. However, in the worst case, when an application request execution privileges for a cache-resident, modified page, MINI-ME forces a page re-fetch from the Page Fault handler to ensure the scanning of the modified page. We highlight that handling SMC is a corner case already affecting existing CPU's performance due to the need of evicting trace cache and stalling pipelines [33], and the MINI-ME's main goal is not to speed up SMC detection, but to prevent imposing overhead to benign, non-SMC applications. For a complete SMC handling, we advocate for the MINI-ME's operation along with an SMC-aware processor [9].

5.3 Matching Engine

MINI-ME's key component is the Matching Engine implemented inside the memory controllers. For the case of smart memories, it is composed by one or many (see experimental results on Section 6) signature database(s) on the logic layer and individual comparison units on each Vault. A similar approach could be used for multiple channel DDR memories. The signature database is a multiple port memory that allows querying for multiple signatures per cycle. The number of ports is tied to the number of smart memory's Vaults and the number of cycles the checks must take. A comprehensive performance and storage evaluation on these numbers is presented in Section 6. The structure of both the database and the comparison units are tied to the selected data storage methods. We have identified distinct implementation possibilities, described below. For the sake of evaluation, we have designed and simulated versions of MINI-ME using all of them.

Direct Mapped Table: When using a direct mapped table, the signature bytes are used to directly index a table entry. The content of such entry is a bit indicating if such signature is malicious (1) or not (0). To include a signature for a newly detected sample, the software-based AV component must only to enable the bit on the corresponding signature index. A drawback of this project decision is that the table exponentially grows with the signature size, becoming prohibitive for large signatures. The practical limits of using a table as the database are discussed in Section 6.

Signature Tree: An alternative for signature storage is to encode the table as a tree, thus each signature byte indexes a distinct table (or table region). Using a tree may reduce the required storage when compression techniques are applied, as non-used indexes/tables may be removed. Updating a hardware database representing a compressed tree is an implementation challenge due to storage constraints, as evaluated and discussed in Section 6.

Bloom Filter (BF): To overcome the exponential storage growth of tables and trees, a probabilistic data structure might be used, so we also implemented a MINI-ME version based in BFs [2]. With it, only some bits are required to represent larger signatures. Although tables are perfect matching structures, BFs may present some False-Positives (FPs), evaluated in practice in Section 6. Despite tables and trees use signature bytes themselves for indexing, a BF requires the use of some hashing functions. All bits used by the hash functions to represent the signatures are stored as a single, large value. Therefore, adding a new signature to a BF database is performed by setting the respective signature bits as present on this long value, as shown in Figure 5.

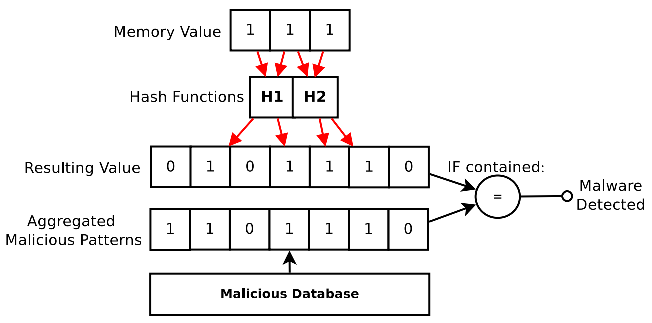

Figure 5: The memory value is hashed into a value which may trigger a detection flag if contained in the aggregated malware signature database.

Table 3: Detection Function. Truth Table

Signature	0	0	1	1
Pattern	0	1	0	1
Detection	1	0	1	1

The detection function depicted in Figure 5 identifies whether a pattern **P** is compatible with the signature **S**, thus possibly triggering a detection flag **D**. The truth table for the detection function is shown in Table 3. On the one hand, if the signature has a given bit set (lines 2 and 3), any pattern might match it. On the other hand, if the signature has a given bit unset (lines 0 and 1), only a pattern with that same bit unset might match it (line 0). Notice that this operation is performed for each bit of S and P. The final detection notification is triggered only if all bits match (all set to 1).

$$D = S \lor \neg P \quad (1) \qquad D = \neg S \not\wedge P \quad (2)$$

The circuit to implement the detection function can be straightforwardly derived from the truth table. It is represented by the

Equation 1. Alternatively, by applying the `De Morgan`'s theorem, it can also be represented by the Equation 2. This implementation is considered more practical because, in practice, MINI-ME does not need to actually negate the signature S using a logic circuit. Instead, the AV company can distribute already-negated signatures.

5.4 Whitelisting memory regions:

MINI-ME implements the whitelist mechanism as a single bit which enables/disables MINI-ME for setting the detection flag for the mis-detected memory address. Once disabled, the scanning procedure is only re-enabled to that memory address after the next memory write on the same location. This mechanism requires MINI-ME to add a control bit to each signature-sized memory region which encompasses the mistakenly matched signature. The relative cost of adding a bit for each word of a given signature size (shown in the Table 4) does not depend on the total RAM capacity, as they are based only in the signature size. Notice that this mechanism does not flag the signature as whitelisted, but the memory region. Therefore, the same signature can be responsible for detecting malware on distinct memory regions.

Table 4: Whitelisting. Storage overhead of adding control bits. The rates are independent of total memory size.

	Signature size		
	32B	**64B**	**128B**
Memory (%)	0,39%	0,20%	0,10%

As a whitelist bit is added to each region corresponding to a word of the same size as a malware signature, distinct signatures sizes will reserve distinct amounts of memory to implement their whitelist bits. More specifically, the larger the signature size, less bits are required to whitelist their regions. When implementing the whitelisting mechanism, we must consider both the required storage space as well as the impact of signature size, to be discussed in Section 6). Such project decisions reflect a trade-off between memory space and processing time, as also existing in most computer science problems. The idea of moving AV from software to hardware eliminates the performance overhead problem (performance gain), but requires additional storage (space impact). Similarly, one can choose to also use additional memory (space impact) to eliminate the performance impact of handling false positives (achieving higher performance).

5.5 Signature generation

Generating good signatures is a crucial step for achieving high detection rates. Traditional AVs rely on sequence of bytes from binary files and moving for memory-based signatures requires paying attention to memory mapping details [60]. When loaded in memory, an executable binary file does not match exactly its disk counterpart. More specifically, for the Windows PE binary case, our focus in this work, Microsoft specifies [60] that the binary `Section Alignment` field specifies: "The alignment (in bytes) of sections when they are loaded into memory. It must be greater than or equal to FileAlignment. The default is the page size for the architecture.". It indicates that the binary file content (distinct binary sections) might not be contiguous when mapped in memory. Therefore, if an ordinary signature procedure is used and the sequence traverse two or more binary file sections which are mapped separately in memory, the signature might be split.

To avoid such effect, distinct approaches might be adopted: (i) Limit signature generation to the code within the same binary section; (ii) Ensure that sections are mapped contiguously in memory; or (iii) Generate Signatures directly from memory images. The first two cases are naturally derived from ordinary AV signature generation procedures, but the third is a new approach. While its adoption is optional for ordinary binaries, it is the only possibility for AV companies to handle fileless malware. Moreover, to define section boundaries, the signature generation procedure must select significant binary sections, such as sections whose content might allow distinguishing a binary from other. Therefore, AV signatures are often implemented based on the `.text` binary section (see Section 3, because sequences of instructions may define a malicious behavior. As an advantage of relying on memory patterns, we may extend the signature generation policy to include any section, which allows matching other patterns, such as strings. As a drawback, the number of false positives may grow if a too comprehensive policy is allowed.

To mitigate FP detection, some known patterns must be avoided. The most significant one is related to the PE header used by the Windows executable files evaluated in this work. As all PE binaries start with the `MZ` string [60], the corresponding hex pattern (`0x4d5a`) should not be used as part of a signature. If so, it will match any other loaded PE binary in the system. Moreover, signatures that matches with well know library functions, such as `printf`, should also be avoided, as currently already done by AV companies.

6 EVALUATION

In this section, we evaluate MINI-ME regarding theoretical (exploratory) and practical aspects. The design exploration is intended to highlight the multiple possibilities enabled by MINI-ME. The practical evaluation aims to show how MINI-ME could be deployed in a near future.

6.1 Exploration: Signature Size

As in the long-term our method ideally traverse the whole memory, signatures must be carefully chosen to reduce the match to non-malicious patterns, which would result in a false positive detection. To mitigate such cases, we need to choose a signature size which reduces the probability of such occurrences. Table 5 shows the results of our experiment using different signature sizes and dumps during the matching. We considered signatures sizes of up to 64 bytes, the current cache line size for most modern processors. We leveraged 100 thousand distinct signatures randomly generated from malicious binaries and matched them against memory dumps of running Windows 7 applications, including the Internet Explorer 10, Firefox 59, Chrome 65, and the 29 applications from the SPEC-CPU 2006 benchmark suite.

The signature sizes of 32 and 64 bytes present no FP with any other pattern from any memory dump, which makes MINI-ME compatible to current AVs: a current AV may use an average of 28 bytes per signature [21] and up to 60KB [22] in the worst case; The whole Clamav database is about 112 MB [14] to store all its million

Table 5: Signature Generation. Signatures (%) detected as false positives for each signature size and memory dump size.

		Memory Size			
		1 GB	2 GB	4 GB	8 GB
Signature Size	8 B	8.65%	9.92%	10.18%	11.45%
	16 B	3.06%	3.32%	3.32%	3.32%
	32 B	0.00%	0.00%	0.00%	0.00%
	64 B	0.00%	0.00%	0.00%	0.00%

signatures. Our 32 and 64-byte-long signatures would require 32 and/or 64 MB, respectively, to store 1 million signatures.

6.2 Exploration: Signature Quality

In addition to effectively detect the malware samples, a good signature must not cause FPs. This imposes an additional requirement to the already-complex AV signature generation procedures.

To understand how to generate good signatures is an important step since bad signatures may lead to false positives. In our tests, we noticed the majority of conflicting signatures presented a pattern of repeated bytes, such as `0x0000`. This may be related to data padding bytes and/or initial values assigned by the memory allocation subsystem. Sequences like `0x9090` also often appears because of they are related to `NOP` sleds, used for instruction padding.

A good policy would be to avoid generating signatures from such patterns. More than avoiding regular patterns, we also suggest avoiding generating signatures from patterns that provide a small amount of information, which may not be suitable for unique identification. As a general metric for such, we suggest using the information entropy [26] concept. Table 6 shows entropy values for some signatures/patterns.

Table 6: Entropy values for distinct signatures. Low values are more probably reported as FPs.

Signature	Entropy	Quality
0x0000000000000000	0.00	✗
0x9090909090909090	1.00	✗
0x5833917ca7fc967c	3.15	✓

The first two signatures were reported as false positives for all memory dumps whereas the third correctly uniquely identified a malware sample. As can be noticed, the entropy value for the third case is much higher than the previous ones. Therefore, a threshold can be used on the signature generation procedure to ensure their quality.

6.3 Exploration: Matching Mechanisms

To determine the FP rates when using distinct matching mechanisms, we have performed an experiment that matches 100 K signatures of malware on a clean machine with 1 GB RAM populated with the execution of the aforementioned benign software. The results are shown in Table 7.

We observe that the two exact matching mechanisms (`Direct Mapped Table` and the `Signature Tree` present the same results whereas the `Bloom Filter` is also affected by FPs due to its intrinsic probabilistic characteristic. FP rates were closer to the ones

Table 7: Matching Techniques. FP rates for multiple signature sizes and techniques.

		Signature size			
		8 B	16 B	32 B	64 B
Match. Tech.	Dir. Mapped Table	8.33%	3.15%	0.00%	0.00%
	Signature Tree	8.33%	3.15%	0.00%	0.00%
	Bloom Filter	8.41%	3.47%	0.00%	0.00%

previously estimated for the smaller signature sizes and no FP was observed for the longer ones even when using BFs, showcasing it as a viable alternative.

6.4 Exploration: Scan Policies

To evaluate different scan policies, we considered the same 100 K signatures and the 1 GB dump. To provide an exact result, we performed this experiment using a Direct Mapped Table as storage for the matching mechanism. The results are shown in Table 8.

Table 8: Scan Policies. FP rate for multiple signature sizes and policies.

		Signature size			
		8 B	16 B	32 B	64 B
Scan Policy	Whole Memory	8.33%	3.15%	0.00%	0.00%
	Mapped Pages	0.06%	0.01%	0.00%	0.00%
	Whitelist	0.00%	0.00%	0.00%	0.00%
	Code-Only	0.01%	0.00%	0.00%	0.00%

We observe that matching the whole memory increases the FP rate. It was expected as looking to more data increases the chance of finding a colliding pattern. Limiting the scan to only the mapped pages significantly reduces FPs, as fewer locations are checked. As an additional restriction, limiting the checks to code regions eliminates the FP which occurred on data pages. However, this approach does not completely eliminate all FPs when using a small signature size. Therefore, larger signatures still present the best results for the general case, achieving no FP at all. As expected, whitelisting previously misdetected regions completely mitigated FPs.

The project decision of the used signature size and matching policy presents another interesting trade-off: by enforcing the use of one of the restricted scan modes, an AV may use smaller signatures, which requires less storage space and makes MINI-ME's definitions updates faster, as fewer bytes will be written to the MINI-ME's control region (although we don't consider this update time as critical as the scan time); On the other hand, it makes the solution less flexible, as it will not be able to operate on a broader threat model, which may require, for instance, to scan all memory pages.

6.5 Exploration: Storage Space Overhead

Once we defined the boundaries for the signature size (32 bytes), we can estimate the impact of implementing the distinct storage strategies.

Static, directly indexed table requires $N_{Signatures} * Size_{Signatures}$ bits to store a pre-defined set of signatures. Therefore, to store 1M 32-byte-long signatures, 32MB of storage is required. Notice that, in the case of a static table, additional, the inclusion of additional signatures are not supported.

Signature trees allows storing signatures in a compressed way. By using an Alphabet Compression Table (ACT) [38], MINI-ME was able to store a pre-defined set of 1M signatures of multiple sizes in a compressed way. Table 9 shows, respectively, the signature size (in bytes), the total size required to sequentially store the signatures on an uncompressed way (without update support) and the total storage space required for the compressed values (without update support).

Table 9: Tree Compression. Larger signatures can be more compressed than smaller ones.

	Signature Size			
	8B	**16B**	**32B**	**64B**
Uncompressed (MB)	8	16	32	64
Compressed (MB)	8	15	16	35

The `Uncompressed` column refers to the size to store the signatures sequentially, being computed as $Number_{Signatures} * Size_{Signatures}$, as for tables, therefore being considered as the basis for comparison (base case).

We observe that the smaller tables were compressed to sizes closer to the base case (uncompressed signatures). The best compression cases are identified on larger signatures, as these present longer sequences of repeated bytes, resulting in more gain. The total storage space required for the 32 and 64 byte-long signatures were closer to 50% of the base case, representing a significant storage gain.

Updates: A compression drawback Whereas we can compress the database tree for an initially defined set of signatures, we cannot guarantee that the database structure will be preserved after signature definition updates. Thus, we need to support reconfigurable hardware or to offer support for the so-called "worst-case", which requires having storage space (thus, hardware) for all combinations in the tree, despite the entries being in use or not. In this case, the tree (or table) representation would require to provide space to store all bits for all signatures, in a total of $2^{Signature-Size}$ bits. Therefore, for the established signature size (32 bytes), MINI-ME would have to store 2^{32*8} bits, which is impractical due to the storage overhead, i.e. required DRAM area. Therefore, the static table and tree representations are more suitable for scenarios that do not require constant database updates. For scenarios of constant updates, the following presented alternatives are better suited.

Bloom filter (BF) We also evaluated MINI-ME's implementation using a BF. The required size to store `n` elements with a FP rate `p` is given by the formula presented in Equation 3. The number of hash functions required to achieve such FP is shown in Equation 4.

$$m = \lceil \frac{n * logp}{\log \frac{1}{2^{\log 2}}} \rceil \quad (3) \qquad k = \lceil \log 2 * \frac{m}{n} \rceil \quad (4)$$

Therefore, based on the defined signature sizes, we can compute the required storage space to implement a bloom filter-based database. Table 10 exemplifies the storage and hash requirements to store 1M signatures for given FP rates.

Similar to previous cases, the BF implementation is backed by a trade-off regarding space and performance. The small the tolerance to FPs (and thus to the overhead of verification routines), more storage space is needed, as more bits will be used. However, even when set to present FP rates closer to zero (less than 0.1%), the total required space is smaller than in the compressed tree, which makes BFs suitable for MINI-ME's implementation in a dynamic scenario.

Table 10: Bloom Filter. FPs and storage space trade-off. The more storage space, less FPs.

	False Positives (1 in N)				
	10	**100**	**1K**	**1M**	**10M**
Hashes (#)	3	7	10	20	23
Storage (MB)	0.58	1.10	1.70	3.40	4.00

For our hypothetical case of storing 1M signatures, a rate of 1 FP in 10M gives results closer to the exact match. In practice, our tests indicated zero FP raised. Therefore, it is considered a good implementation choice.

Finally, we highlight that the number of required hash functions do not impose any constraint to MINI-ME implementation, as they can be implemented as independent, parallel bitwise functions within the smart memories' controllers.

6.6 Practice: Database Size Definition

MINI-ME's goal is not to move the whole AV detection capabilities from software to memory, but only the engine components responsible for fileless malware detection. Therefore, MINI-ME does not need to support all 1M signatures supported by the software AV, but only the challenging ones, i.e., the ones responsible to detect malware samples that can only be detected in runtime. In this sense, although the number of fileless malware samples has grown 94% in the last years, they are currently responsible for only 4% of all attacks.

Moreover, AVs will not deploy signatures to all known fileless malware samples ever existing, but only to the active ones in a given period of time. In this sense, despite harmful, fileless malware samples are still limited in number, with only (the same) one present in the list of most active malware samples of 2018 [64] and 2019 [65].

Therefore, we limited MINI-ME's current signature database to only 1K entries to benefit from smaller energy and area costs. A bloom filter to store 1K entries with 0.1% FP requires only 1.7KB of space per Vault. Since HMCs have a maximum number of 32 Vaults [18], thus memory controllers, MINI-ME currently requires less than 64KB of memory to support an entire HMC memory. The storage capacity might be increased over time as fileless samples become more popular.

6.7 Practice: Database Implementation

The previously presented calculation defined that MINI-ME requires 1.7KB of memory/per Vault to implement its database. Ideally, this memory should be as fast as possible to reduce the imposed overhead. Registers are suitable candidates to meet this requirement. However, the largest registers currently available on modern platforms are the 512-bit long Intel AVX2 registers [32], which requires MINI-ME to split its match routine across multiple cycles if operating with AVX-2 like registers. Notice that MINI-ME requires AVX2-sized registers but it does not need to implement AVX's complex, associated control mechanisms because MINI-ME does not implement vector operations.

The number of cycles required by MINI-ME to perform its match depends on the number of available registers and how fast these can be accessed and compared. In our tests, we consider a conservative scenario in which only one AVX2-like register is available, but multiple checks can be performed in parallel if more registers are available. We also consider that each comparison takes a cycle, assuming that the reference register is previously loaded with the fixed malware database. Finally, we considered that each match was performed until the end of the matching pattern, with no optimization. Notice that, in practice, the matching routine might stop after the first unmatched pattern. The remaining cycles could be used, for instance, to perform checks on unaligned patterns (see Section 7). Considering this conservative scenario, MINI-ME required ≈32 cycles (1.7KB/512bits) per check.

We evaluate the imposed overhead imposed by MINI-ME in multiple scenarios by simulating a memory controller that imposes distinct delays to write requests. The simulation was performed on a cycle-accurate simulator that emulates internal structures of an HMC-powered x86 processor [4]. We considered the applications from the SPEC benchmark, as in Section 2. All traces were composed of 200M instructions extracted by Intel pin [40] while using the pinpoints method [59].

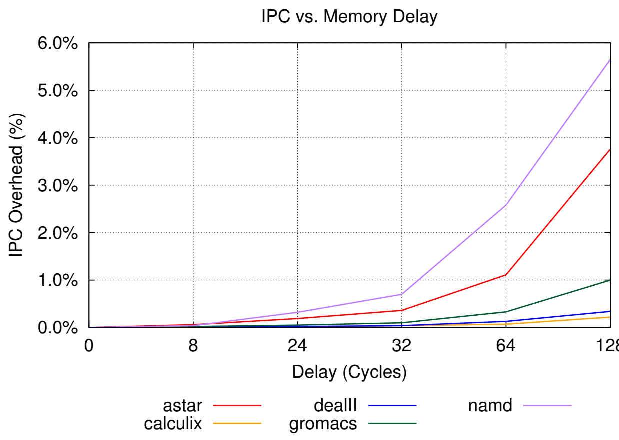

Figure 6: MINI-ME database overhead. Delays of up 32 cycles impose less than 1% of IPC overhead.

Figure 6 shows the imposed overhead in terms of Instructions Per Cycle (IPC) penalty for different memory delays (in cycles). Despite showing only some benchmark applications, the results hold for all applications. Memory delays up to 32 impose IPC overheads smaller than 1%, thus not significantly affecting overall system performance while increasing security coverage.

In the long-term, if adding a huge number of fileless malware signatures become a requirement, the memory delay might be increased to support larger databases. Longer memory delays impose significantly greater overhead, of up to 5%. However, we still consider this trade-off reasonable, since the same malware detection approach imposes overheads of up to 100% when implemented in software (see Section 2).

Energy Efficiency As for access time, the required storage also reflects a trade-off regarding the energy costs and the system performance. MINI-ME adds a database to the whole system and requires each Vault to have an additional register in constant operation. As for the previous case, we consider this trade-off acceptable as the implementation of the same malware detector in software would cause a higher energy consumption due to the need of polling. The chip area to implement MINI-ME is entirely dominated by the SRAM. Therefore, the area and energy costs are directly proportional to the number of available registers.

6.8 Practice: Monitoring Overhead

Once we have defined MINI-ME parameters, we aimed to evaluate its performance in practice when configured with them. However, performing a fair comparison to existing commercial solutions is hard because we do not have access to all the parameters leveraged by the closed source solutions (e.g., accessed pages, signature size, policies). Therefore, we opted to compare MINI-ME against an academically-proposed memory inspector [1], since its parameters are available. On the one hand, its detection capability might not be as good as a commercial AV because it is not a full fileless malware detector, but checks only a subset of the memory-related API calls when these are invoked by any application (on-access inspection). On the other hand, this solution is a very lightweight approach and thus can highlight how MINI-ME is effective in mitigating overhead even face to a lightweight solution. As no source-code was available, we re-implemented the solution according to our understanding of what would be done by an AV company. We considered the same APIs described in the paper and instrumented them via userland hooks [5]. The proposed solution hashes memory data using the MD5 algorithm. We considered the MS implementation [49] for this task so as to benefit from its optimized performance.

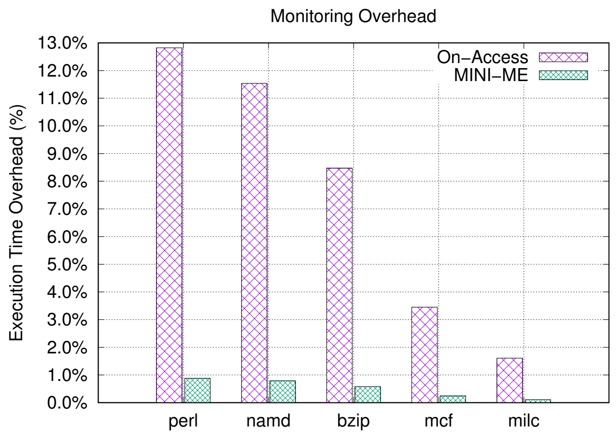

Figure 7: Monitoring Overhead. MINI-ME imposes a smaller overhead while still checking more pages than an on-access solution.

Figure 7 shows the execution time overhead from applying the on-access method and MINI-ME over the same SPEC applications presented in Section 2. This only accounts for the monitoring step and not for the notification message deliver (e.g., I/Os) nor the application of post-detection procedures (e.g., process blocking. We notice that although the overhead of this lightweight approach is significantly small in practice than the worst-case discussed in Section 2, it is still significant for most applications. In addition, this result might be even worse if more comprehensive checks are performed by non-lightweight monitoring solutions. In turn, MINI-ME imposed a negligible overhead to all applications (in no case, the overhead was greater than 1%) even performing much more comprehensive checks than the lightweight approach. This shows that MINI-ME is a promising solution for overhead mitigation in fileless malware detection procedures.

6.9 Practice: Malware Detection

To evaluate MINI-ME in practice, we considered the execution of 21 thousand real malware samples collected over four years. We had access to this dataset that has already been characterized by a previous work [12] and proven to be challenging to other classification tasks [7]. We considered a database of up to 1K signatures, composed of a sequence of 32 random bytes from the `.text` binary section, on a 1GB memory, populated with the execution of the benign applications previously described. The match was performed against the whole memory using the BF mechanism. In each experiment round, we added 500 signatures of malware samples that were loaded in memory and 500 signatures of malware that were not loaded in the system's memory. MINI-ME matched the signatures for all samples without triggering FPs, demonstrating that MINI-ME is practical in real scenarios.

7 DISCUSSION

In this section, we revisit our contributions to discuss their implications and the limits of MINI-ME application.

MINI-ME advances MINI-ME provides a platform for AV instrumentation with negligible overhead, allowing them to perform constant whole-memory checks, a limitation of the existing models. Reducing the performance impact of memory scans for fileless malware detection allows MINI-ME to mitigate the impact of this type of threat in actual systems. MINI-ME is a practical solution as it does not cause a paradigm shift in detection techniques, but leverages existing AV industry knowledge for threat detection. AV companies may still develop their investigations and distribute customized signatures, which makes MINI-ME an easy-to-adopt approach. MINI-ME is also transparent for applications, thus not requiring any code injection, recompilation nor introducing side effects. Therefore, all existing applications would be protected when running in a MINI-ME-powered environment.

Good detection rates depend upon good policies. MINI-ME provides a platform for efficient signature matching. However, the detection effectiveness is still depending upon the security policies defined by the AV company. MINI-ME's flexibility allows, for instance, AV companies to opt for the use of both a single or multiple signatures for the same malware samples, thus allowing them to control their confidence on the reported detection. This way, MINI-ME does not break the current AV market, as it still allows AV companies to offer customized services according to their customer's needs. MINI-ME is also flexible to allow AV companies to split large signatures into multiple smaller signatures to be matched by the hardware component. In this case, the AV companies might further reorder and rebuild large stream in the userland component handling the detection notifications.

Matching Unaligned Patterns. A detection policy decision that let for AV companies is about matching unaligned signatures. In this work we assumed that our signatures were all aligned, which was true for all of our experiments. However, AV companies might identify that a given pattern might be revealed in multiple, distinct locations during a fileless malware execution and thus opt to detect the threat via this pattern. MINI-ME might support this type of scan by allowing signature matching procedures to occur at distinct offsets of the scanned data packets. The scanning procedure might be repeated until the selected number of strides defined by the AV company and the matching pattern is disambiguate at software-level by the intelligence agent implemented by the AV. The performance of this approach is dependent on the number of strides and the number of matching registers available on the memory controller. In the first case, the greater the number of strides. the greater the overhead, In the second case, the more registers are available for parallel checks, the faster the match. MINI-ME is not able to detect patterns that are split across the boundaries of a write request (the cache line size).

Transition to Practice. The proposed approach of performing pattern matching during the write-to-read window can be implemented both in a near-memory manner (extending the memory controller inside the CPU) and in a in-memory manner (extending the memory controller inside smart-memories). ISA modifications are not required in any case. Whereas implementing MINI-ME in a near-memory manner is straightforward for current architectures, in this work, we implemented MINI-ME prototype in an in-memory manner already envisioning MINI-ME transition to future operational scenarios, even though they are more challenging due to the timing constraints imposed by smart-memories. Finally, during all MINI-ME development process, simplicity was envisioned as a key target, thus making MINI-ME fully portable to many architectures and platforms.

Limitations MINI-ME requires AV companies to generate new signatures for each newly discovered malware variants. **We do not consider it as a particular MINI-ME limitation** because it is a drawback for current AVs and affects all signature-based solutions [51, 53]. Therefore, handling malware variants is out of MINI-ME's scope. MINI-ME's choice by a signature-based approach is supported by its widely adoption to detect fileless malware, as shown in Section 3. In this sense, MINI-ME also do not handle new malware samples created by misaligning previously identified signatures, as these are considered as malware variants by already existing AV solutions.

Future of fileless malware Attackers exploit gaps and fileless malware is a clear example of it. Such threat is hard to be detected by AVs, either by performance constraints or by infecting legitimate processes. MINI-ME raises the bar for such exploitation, so attackers shall move to exploit other gaps. We believe that, with MINI-ME adoption, attackers will follow the same steps took in ordinary samples evolution, such as applying polymorphism to hide their signatures from AVs. Similarly, AVs will evolve to flex their signature schema to handle such cases. Therefore, we envision MINI-ME as the first step of hardware-assisted support for malware detection and expect other researchers to benefit from our framework to react to future threats. The next-generation MINI-ME would be probably required to support regular-expression-based matching, which imposes a significant development challenge, as it requires storing arbitrary-size regex automatas in a constrained memory database, which will be considered in future work. We also believe that defensive measures should not to be only reactive, but also proactive, thus legitimate software must properly protect themselves to avoid being infected and leveraged to threat their users, thus also contributing to fight payload injection by fileless malware.

From Signatures to Regex and Machine Learning. As far as we know, MINI-ME is the first solution relying on an in-memory/near-memory mechanism for malware detection. Therefore, it should be understood as a platform for future developments of hardware-assisted AVs. Although evaluated using signatures, the concept proposed by MINI-ME can be leveraged to support any other detection mechanism that can be implemented in hardware, such as a port of the ClamAV [56] to match regular expressions, or the use of ML algorithms to identify malicious memory accesses patterns [6]. These algorithms, however, are more processing-demanding than signature matching. If they require a significant number of cycles to be processed, they might not immediately benefit from the write-to-read window explored in our solution. Thus, detection solutions based on these approaches should consider the adoption of co-processors and/or FPGAs, as suggested by previous studies [58].

Beyond MINI-ME Despite being focused on malware detection, MINI-ME may be employed on distinct scenarios that requires more efficient pattern matching approaches, such as for rootkit detection. A typical rootkit strategy is to hide processes by removing them from the kernel list [30]. Our mechanism, however, can identify running processes by their signatures, regardless of kernel information. In a summary, the pattern matching detection of malware samples can be considered as a particular case of a generalized pattern matching procedure, as it imposes tighter corner cases. As an example, benign programs often present well defined magic numbers which do not match other memory values, thus not requiring whitelisting. Therefore, our approach can be used for general pattern matching without modifications, since the benign program match uses laxer conditions than the ones we presented in this paper. Finally, although focused on smart DRAM memories, MINI-ME can be extended to other memory architectures. As future work, we will investigate how to perform pattern matching on memristor-based systems.

8 RELATED WORK

In this section, we present related work to better position our contributions.

Fileless Malware. Our work is motivated by the detection of a sample through a memory pattern matching that identified the presence of the code from the `Meterpreter` exploitation framework [66] inside a process memory [19]. The malware movement towards memory-based implementations and the need of performing whole-memory pattern matching to detect them—which is costly—pointed us the need of developing better memory pattern matching mechanism to detect future threats. A comprehensive description of fileless malware operation is presented by Sudhakar and Kumar [70].

Hardware AVs. Previous work on efficient malware detection have suggested implementing hardware-assisted AVs in FPGAs [27], which present many drawbacks to be implemented in actual systems. Ho and Lemieux [29] proposed moving ClamAV signatures and regular expressions to an FPGA. Their solution, however, is limited to a immutable signature database, not being suited to be used with dynamic AV signature definitions. Lin et al. [39] presented a bloom filter-based matching solution. They use a constrained storage table which is limited to 10 thousand distinct signatures, with no updates. In addition, their FPGA implementation limits the solution to work as a co-processor, and not as a fully integrated mechanism. Due to these limitations, smart memories were considered good candidates for implementing MINI-ME.

Processing In-Memory (PIM). The PIM feature allows MINI-ME to be implemented as a fully integrated security mechanism. In fact, adding processing capabilities to DRAM presents high potential of overhead elimination for many operations, such as supporting vector operations [3], query processing on big-data databases [63], and for neural networks implementation [54]. Despite the PIM research growth, as far as we know, no other work has proposed to move AV and matching procedures to the memory controller of smart-memories. MINI-ME also relates to the in-disk processing concept [61], in which processing capabilities are added to hard disks. MINI-ME could be ported for such devices since they rely on large buffers and have a logic controller which can be instrumented to perform pattern matching operations.

9 CONCLUSIONS

We investigated the problem of real-time, memory scanning for fileless malware detection and proposed near-memory and in-memory approaches to perform malicious signature-matching during the write-to-read time window, thus eliminating the performance penalty of polling routines implemented by software-based AV solutions. As a proof of concept, we developed MINI-ME (Malware Identifier by Near- and In-Memory Evaluation), an in-memory, AV hardware accelerator able to perform continuous memory scans to match signatures at new data writes to the main memory and notifying a traditional software-based AV when a signature is found. MINI-ME implementation was made practical via the use of bloom filters to reduce the storage size of 1M signatures to only 4MB and to allow pattern matching to be performed within the DRAM buffers even when a write request is followed by a read request in the same open cells (e.g., CAS time). Experimental results showed that MINI-ME was able to detect multiple sets of 500 real-world malicious samples each with zero overhead and no FPs, thus demonstrating its viability.

Reproduciblity. The developed prototype's source code is available at: https://github.com/marcusbotacin/In.Memory

ACKNOWLEDGMENTS

This project was partially financed by the Serrapilheira Institute (grant number Serra-1709-16621) and by the Brazilian National

Counsel of Technological and Scientific Development (CNPq, PhD Scholarship, process 164745/2017-3).

REFERENCES

[1] Mohammed I. Al-Saleh and Rasha K. Al-Huthaifi. 2017. On Improving Antivirus Scanning Engines: Memory On-Access Scanner. https://thescipub.com/abstract/10.3844/jcssp.2017.290.300. *Journal of Computer Sciences* 13, Article 1 (2017), 10 pages.

[2] Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. 2007. Scalable Bloom Filters. *Inf. Process. Lett.* 101, 6, Article 1 (March 2007), 7 pages. https://doi.org/10.1016/j.ipl.2006.10.007

[3] M. A. Z. Alves, M. Diener, P. C. Santos, and L. Carro. 2016. Large vector extensions inside the HMC. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*. UEEE, US, 1249–1254.

[4] M. A. Z. Alves, C. Villavieja, M. Diener, F. B. Moreira, and P. O. A. Navaux. 2015. SiNUCA: A Validated Micro-Architecture Simulator. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. UEEE, US, 605–610. https://doi.org/10.1109/HPCC-CSS-ICESS.2015.166

[5] apriorit. 2018. A Windows API hooking library. https://github.com/apriorit/mhook.

[6] Sergii Banin and Geir Olav Dyrkolbotn. 2018. Multinomial malware classification via low-level features. *Digital Investigation* 26 (2018), S107 – S117. https://doi.org/10.1016/j.diin.2018.04.019

[7] Tamy Beppler, Marcus Botacin, Fabrício J. O. Ceschin, Luiz E. S. Oliveira, and André Grégio. 2019. L(a)ying in (Test)Bed. In *Information Security*, Zhiqiang Lin, Charalampos Papamanthou, and Michalis Polychronakis (Eds.). Springer International Publishing, Cham, 381–401.

[8] Marcus Botacin, Paulo Lício De Geus, and André Grégio. 2018. Enhancing Branch Monitoring for Security Purposes: From Control Flow Integrity to Malware Analysis and Debugging. *ACM Trans. Priv. Secur.* 21, 1, Article 4 (Jan. 2018), 30 pages. https://doi.org/10.1145/3152162

[9] Marcus Botacin, Marco Zanata, and André Grégio. 2020. The self modifying code (SMC)-aware processor (SAP): a security look on architectural impact and support. *Journal of Computer Virology and Hacking Techniques* 1, 1 (2020), 1. https://doi.org/10.1007/s11416-020-00348-w

[10] Rodrigo Rubira Branco, Gabriel Negreira Barbosa, and Pedro Drimel Neto. 2012. Scientific but Not Academical Overview of Malware Anti-Debugging, Anti-Disassembly and Anti-VM Technologies. https://media.blackhat.com/bh-us-12/Briefings/Branco/BH_US_12_Branco_Scientific_Academic_WP.pdf.

[11] Niklas Broberg, Andreas Farre, and Josef Svenningsson. 2004. Regular Expression Patterns. *SIGPLAN Not.* 39, 9, Article 1 (Sept. 2004), 12 pages. https://doi.org/10.1145/1016848.1016863

[12] F. Ceschin, F. Pinage, M. Castilho, D. Menotti, L. S. Oliveira, and A. Gregio. 2018. The Need for Speed: An Analysis of Brazilian Malware Classifers. *IEEE Security & Privacy* 16, 6 (Nov.-Dec. 2018), 31–41. https://doi.org/10.1109/MSEC.2018.2875369

[13] Mahinthan Chandramohan, Hee Beng Kuan Tan, Lionel C. Briand, Lwin Khin Shar, and Bindu Madhavi Padmanabhuni. 2013. A Scalable Approach for Malware Detection Through Bounded Feature Space Behavior Modeling. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering* (Silicon Valley, CA, USA) *(ASE'13)*. IEEE Press, Piscataway, NJ, USA, Article 1, 11 pages. https://doi.org/10.1109/ASE.2013.6693090

[14] Clamav. 2018. Clamav. https://www.clamav.net/downloads#collapseCVD.

[15] ClamSentinel. 2018. ClamSentinel. https://sourceforge.net/projects/clamsentinel/.

[16] ClamWin. 2018. Free Antivirus for Windows. http://www.clamwin.com/.

[17] Fred Cohen. 1984. Computer Viruses - Theory and Experiments. http://web.eecs.umich.edu/~aprakash/eecs588/handouts/cohen-viruses.html.

[18] Hybrid Memory Cube Consortium. 2013. Hybrid Memory Cube Specification Rev. 2.0. http://www.hybridmemorycube.org.

[19] Cyberscoop. 2017. New malware works only in memory, leaves no trace. https://www.cyberscoop.com/kaspersky-fileless-malware-memory-attribution-detection/.

[20] DarkReading. 2016. Fileless Malware Takes 2016 By Storm. https://www.darkreading.com/vulnerabilities---threats/fileless-malware-takes-2016-by-storm/d/d-id/1327796.

[21] EMSISOFT. 2015. Why antivirus uses so much RAM – And why that is actually a good thing! https://blog.emsisoft.com/2016/04/13/why-antivirus-uses-so-much-ram-and-why-that-is-actually-a-good-thing/.

[22] ESET. 2018. Types of updates. http://support.eset.com/kb309/?viewlocale=en_US.

[23] Facebook. 2018. OSQuery. https://osquery.io/schema/3.3.2.

[24] glmcdona. 2017. Process-Dump. https://github.com/glmcdona/Process-Dump.

[25] Google. 2017. Rekall. https://github.com/google/rekall.

[26] Robert M. Gray. 2011. *Entropy and Information Theory*. Springer, US. https://doi.org/10.1007/978-1-4419-7970-4

[27] N. B. Guinde and R. B. Lohani. 2011. FPGA Based Approach for Signature Based Antivirus Applications. In *Proceedings of the International Conference &#38; Workshop on Emerging Trends in Technology* (Mumbai, Maharashtra, India) *(ICWET '11)*. ACM, New York, NY, USA, Article 1, 2 pages. https://doi.org/10.1145/1980022.1980300

[28] Peter Gutmann. 2007. The Commercial Malware Industry. https://www.cs.auckland.ac.nz/~pgut001/pubs/malware_biz.pdf.

[29] Johnny Tsung Lin Ho and Guy G.F. Lemieux. 2009. PERG-Rx: A Hardware Pattern-matching Engine Supporting Limited Regular Expressions. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '09)*. ACM, New York, NY, USA, Article 1, 4 pages. https://doi.org/10.1145/1508128.1508171

[30] Greg Hoglund and Jamie Butler. 2005. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, US.

[31] F. H. Hsu, M. H. Wu, C. K. Tso, C. H. Hsu, and C. W. Chen. 2012. Antivirus Software Shield Against Antivirus Terminators. *IEEE Transactions on Information Forensics and Security* 7, 5 (Oct 2012), 1439–1447. https://doi.org/10.1109/TIFS.2012.2206028

[32] Intel. 2011. *Intel(R) Advanced Vector Extensions Programming Reference*. Intel.

[33] Intel. 2013. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel.

[34] Bruce Jacob, Spencer Ng, and David Wang. 2007. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[35] Aamer Jaleel. 2012. Memory Characterization of Workloads Using Instrumentation-Driven Simulation. http://www.jaleels.org/ajaleel/publications/SPECanalysis.pdf.

[36] Kaspersky. 2015. Overall Statistics for 2015. https://securelist.com/files/2015/12/KSB_2015_Statistics_FINAL_EN.pdf. Access in May 11, 2016.

[37] Kaspersky. 2017. A Disembodied Threat. https://www.kaspersky.com/blog/bodiless-threat/6128/.

[38] Shijin Kong, Randy Smith, and Cristian Estan. 2008. Efficient Signature Matching with Multiple Alphabet Compression Tables. In *Proceedings of the 4th International Conference on Security and Privacy in Communication Netowrks* (Istanbul, Turkey) *(SecureComm '08)*. ACM, New York, NY, USA, Article 1, 10 pages. https://doi.org/10.1145/1460877.1460879

[39] P. C. Lin, Y. D. Lin, Y. C. Lai, Y. J. Zheng, and T. H. Lee. 2009. Realizing a Sub-Linear Time String-Matching Algorithm With a Hardware Accelerator Using Bloom Filters. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 17, 8 (Aug 2009), 1008–1020. https://doi.org/10.1109/TVLSI.2008.2012011

[40] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. ACM, New York, NY, USA, Article 1, 11 pages. https://doi.org/10.1145/1065010.1065034

[41] Lorenzo Martignoni, Aristide Fattori, Roberto Paleari, and Lorenzo Cavallaro. 2010. Live and Trustworthy Forensic Analysis of Commodity Production Syst.. In *Proc. 13th Intl. Conf. on Recent Advances in Intrusion Detection (RAID'10)*. Springer-Verlag, US.

[42] Micron. 2018. Hybrid Memory Cube – HMC Gen2. https://www.micron.com/~/media/documents/products/data-sheet/hmc/gen2/hmc_gen2.pdf.

[43] Microsoft. 2017. Enumerating All Processes. https://msdn.microsoft.com/pt-br/library/windows/desktop/ms682623(v=vs.85).aspx.

[44] Microsoft. 2017. OpenProcess function. https://msdn.microsoft.com/en-us/library/windows/desktop/ms684320(v=vs.85).aspx.

[45] Microsoft. 2017. ReadProcessMemory function. https://msdn.microsoft.com/pt-br/library/windows/desktop/ms680553(v=vs.85).aspx.

[46] Microsoft. 2018. Getting started with Windows drivers. https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/.

[47] Microsoft. 2018. IsDebuggerPresent function. https://msdn.microsoft.com/en-us/library/windows/desktop/ms680345(v=vs.85).aspx.

[48] Microsoft. 2018. Overview of memory dump file options for Windows. https://support.microsoft.com/en-us/help/254649/overview-of-memory-dump-file-options-for-windows.

[49] Microsoft. 2019. MD5 Class. https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.md5?view=netframework-4.8.

[50] Hyungon Moon, Hojoon Lee, Jihoon Lee, Kihwan Kim, Yunheung Paek, and Brent Byunghoon Kang. 2012. Vigilare: Toward Snoop-based Kernel Integrity Monitor. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (Raleigh, North Carolina, USA) *(CCS '12)*. ACM, New York, NY, USA, Article 1, 10 pages. https://doi.org/10.1145/2382196.2382202

[51] A. Moser, C. Kruegel, and E. Kirda. 2007. Limits of Static Analysis for Malware Detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. ACM, US, 421–430. https://doi.org/10.1109/ACSAC.2007.21

[52] Netmarketshare. 2018. Operating System Market Share. https://www.netmarketshare.com/operating-system-market-share.aspx.

[53] P. OKane, S. Sezer, and K. McLaughlin. 2011. Obfuscation: The Hidden Malware. *IEEE Security Privacy* 9, 5 (Sept 2011), 41–47. https://doi.org/10.1109/MSP.2011.98

[54] Geraldo F. Oliveira, Paulo C. Santos, Marco A. Z. Alves, and Luigi Carro. 2017. NIM: An HMC-Based Machine for Neuron Computation. In *Applied Reconfigurable Computing*, Stephan Wong, Antonio Carlos Beck, Koen Bertels, and Luigi Carro (Eds.). Springer International Publishing, Cham, 28–35.

[55] J. Van Olmen, A. Mercha, G. Katti, C. Huyghebaert, J. Van Aelst, E. Seppala, Z. Chao, S. Armini, J. Vaes, R. C. Teixeira, M. Van Cauwenberghe, P. Verdonck, K. Verhemeldonck, A. Jourdain, W. Ruythooren, M. de Potter de ten Broeck, A. Opdebeeck, T. Chiarella, B. Parvais, I. Debusschere, T. Y. Hoffmann, B. De Wachter, W. Dehaene, M. Stucchi, M. Rakowski, P. Soussan, R. Cartuyvels, E. Beyne, S. Biesemans, and B. Swinnen. 2008. 3D stacked IC demonstration using a through Silicon Via First approach. In *2008 IEEE International Electron Devices Meeting*. IEEE, US, 1–4. https://doi.org/10.1109/IEDM.2008.4796763

[56] N. L. Or, X. Wang, and D. Pao. 2016. MEMORY-Based Hardware Architectures to Detect ClamAV Virus Signatures with Restricted Regular Expression Features. *IEEE Trans. Comput.* 65, 4 (April 2016), 1225–1238. https://doi.org/10.1109/TC.2015.2439274

[57] OSForensics. 2018. OSForensics. https://www.osforensics.com/.

[58] Nisarg Patel, Avesta Sasan, and Houman Homayoun. 2017. Analyzing Hardware Based Malware Detectors. In *Proceedings of the 54th Annual Design Automation Conference 2017* (Austin, TX, USA) *(DAC '17)*. Association for Computing Machinery, New York, NY, USA, Article 25, 6 pages. https://doi.org/10.1145/3061639.3062202

[59] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. 2004. Pinpointing Representative Portions of Large Intel ® Itanium ® Programs with Dynamic Instrumentation. In *37th International Symposium on Microarchitecture (MICRO-37'04)*. ACM/IEEE, US, 81–92. https://doi.org/10.1109/MICRO.2004.28

[60] Matt Pietrek. 1994. Peering Inside the PE: A Tour of the Win32 Portable Executable File Format. https://msdn.microsoft.com/en-us/library/ms809762.aspx.

[61] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. 2001. Active Disks for Large-Scale Data Processing. *Computer* 34, 6, Article 1 (June 2001), 7 pages. https://doi.org/10.1109/2.928624

[62] C. Rossow, C. J. Dietrich, C. Grier, C. Kreibich, V. Paxson, N. Pohlmann, H. Bos, and M. v. Steen. 2012. Prudent Practices for Designing Malware Experiments: Status Quo and Outlook. In *2012 IEEE Symposium on Security and Privacy*. IEEE, US, 65–79. https://doi.org/10.1109/SP.2012.14

[63] P. C. Santos, G. F. Oliveira, D. G. Tomé, M. A. Z. Alves, E. C. Almeida, and L. Carro. 2017. Operand size reconfiguration for big data processing in memory. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*. IEEE, US, 710–715. https://doi.org/10.23919/DATE.2017.7927081

[64] Cis Security. 2018. Top 10 Malware January 2018. https://www.cisecurity.org/blog/top-10-malware-january-2018/.

[65] Cis Security. 2019. Top 10 Malware January 2019. https://www.cisecurity.org/blog/top-10-malware-january-2019/.

[66] Offensive Security. 2017. Using Meterpreter Commands. https://www.offensive-security.com/metasploit-unleashed/meterpreter-basics/.

[67] Sarabjeet Singh and Manu Awasthi. 2019. Memory Centric Characterization and Analysis of SPEC CPU2017 Suite. https://www.cs.utah.edu/~manua/pubs/icpe19a.pdf.

[68] Nae Young Song, Yongseok Son, Hyuck Han, and Heon Young Yeom. 2016. Efficient Memory-Mapped I/O on Fast Storage Device. *ACM Trans. Storage* 12, 4, Article 19 (May 2016), 27 pages. https://doi.org/10.1145/2846100

[69] SPEC. 2006. CPU 2006. https://www.spec.org/cpu2006/. This suite has been retired during the paper development process.

[70] Sudhakar and Sushil Kumar. 2020. An emerging threat Fileless malware: a survey and research challenges. *Cybersecurity* 3, 1 (14 Jan 2020), 1. https://doi.org/10.1186/s42400-019-0043-x

[71] Vasilis G. Tasiopoulos and Sokratis K. Katsikas. 2014. Bypassing Antivirus Detection with Encryption. In *Proceedings of the 18th Panhellenic Conference on Informatics* (Athens, Greece) *(PCI '14)*. ACM, New York, NY, USA, Article 16, 2 pages. https://doi.org/10.1145/2645791.2645857

[72] TechRadar. 2018. Ransomware attacks see huge year-on-year rise. https://www.techradar.com/news/ransomware-attacks-see-huge-year-on-year-rise.

[73] TrendMicro. 2017. A Look at JS_POWMET, a Completely Fileless Malware. http://blog.trendmicro.com/trendlabs-security-intelligence/look-js_powmet-completely-fileless-malware/.

[74] Wired. 2017. Say Hello to the Super-Stealthy Malware That's Going Mainstream. https://www.wired.com/2017/02/say-hello-super-stealthy-malware-thats-going-mainstream/.

[75] Christian Wressnegger, Kevin Freeman, Fabian Yamaguchi, and Konrad Rieck. 2017. Automatically Inferring Malware Signatures for Anti-Virus Assisted Attacks. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (Abu Dhabi, United Arab Emirates) *(ASIA CCS '17)*. ACM, New York, NY, USA, Article 1, 12 pages. https://doi.org/10.1145/3052973.3053002