Hypervisor-assisted Atomic Memory Acquisition in Modern Systems

Michael Kiperberg[1], Roee Leon[2], Amit Resh[3], Asaf Algawi[2] and Nezer Zaidenberg[4]
[1]*Faculty of Sciences, Holon Institute of Technology, Israel*
[2]*Department of Mathematical IT, University of Jyväskylä, Finland*
[3]*School of Computer Engineering, Shenkar College of Engineering, Design and Art, Israel*
[4]*School of Computer Sciences, The College of Management, Academic Studies, Israel*
michaelkip@hit.ac.il, roee.leonn@gmail.com, amitr44@gmail.com, nzaidenberg@me.com

Keywords: Live Forensics, Memory Forensics, Memory Acquisition, Virtualization, Reliability, Atomicity, Integrity of a Memory Snapshot, Forensic Soundness.

Abstract: Reliable memory acquisition is essential to forensic analysis of a cyber-crime. Various methods of memory acquisition have been proposed, ranging from tools based on a dedicated hardware to software only solutions. Recently, a hypervisor-based method for memory acquisition was proposed (Qi et al., 2017; Martignoni et al., 2010). This method obtains a reliable (atomic) memory image of a running system. The method achieves this by making all memory pages non-writable until they are copied to the memory image, thus preventing uncontrolled modification of these pages. Unfortunately, the proposed method has two deficiencies: (1) the method does not support multiprocessing and (2) the method does not support modern operating systems featuring address space layout randomization (ASLR). We describe a hypervisor-based memory acquisition method that solves the two aforementioned deficiencies. We analyze the memory usage and performance of the proposed method.

1 INTRODUCTION

Nowadays, the sophistication level of cyber-attacks makes it almost impossible to analyze them statically. Many of the attacks are designed to detect debuggers and other tools of dynamic analysis. Upon detection of such tool, the malicious software deviates from its normal behavior, thus rendering the analysis useless. Therefore, usually the analysis of an attack is divided into two steps: memory acquisition and static analysis. In the first step, a software (Qi et al., 2017; Martignoni et al., 2010; Reina et al., 2012) or a hardware tool (Zhang et al., 2010) acquires the memory contents of a running system and stores it for later analysis. In the second step, a static analysis tool, e.g., Rekall (Cohen, 2014), is applied to the acquired image of memory to analyze the malicious software.

The memory acquisition is performed while the system is running and updating its data structures and pointers. Consider the following example. The operating system creates a new Process Environment Block at page 1,000 and adds it to the list of running processes at page 2,000. Assume that the process creating occurs when the first 1,500 pages were already acquired. In the resulting memory image, we will have a list of processes that point to an invalid Process Environment Block because page 1,000 was acquired before the creation of the Process Environment Block. Therefore, special measures must be taken to avoid inconsistencies in the acquired memory image.

This paper presents a software hypervisor-based tool for consistent memory acquisition. Hypervisor's ability to configure access rights of memory pages can be used to solve the problem of inconsistencies as follows:

1. When the hypervisor is requested to start memory acquisition, it configures all memory pages to be non-writable.
2. When an attempt is made to write to a memory page P, the hypervisor is notified.
3. The hypervisor copies the contents of P to its inner buffer and configures the page P to be writable.
4. The hypervisor periodically sends the data in its inner buffer. If more data can be sent than is available in the inner buffer, then the hypervisor sends other pages and configures them to be writable.

This method is described in multiple previous works (Qi et al., 2017; Martignoni et al., 2010).

Unfortunately, two problems arise with the described method in modern systems. The first problem is the availability of multiple processors. Each processor has direct access to the main memory and can freely modify any page. Therefore, when the hypervisor is requested to start memory acquisition, it must configure all memory pages *on all processors* to be non-writable.

Another problem is delay sensitivity of some memory pages. Generally, interrupt service routines react to interrupts in two steps: they register the occurrence of an interrupt and acknowledge the device that the interrupt was serviced. The acknowledgement must be received in a timely manner; therefore, the registration of an interrupt occurrence, which involves writing to a memory page, must not be intercepted by a hypervisor, i.e., these pages must remain writable.

Address space layout randomization, a security feature employed by modern operating systems, e.g., Windows 10, complicates the delay sensitivity problem even more. When ASLR is enabled, the operating system splits its virtual address space into regions. Then, during the initialization of the operating system, each region is assigned a random virtual address. With ASLR, the location of the delay-sensitive pages is not known in advance.

We propose the following solution to the problems mentioned above. Our hypervisor invokes an operating system's mechanism to perform an atomic access rights configuration on all the processors. Section 4.3 describes the invocation process, which allows our hypervisor to call an operating system's function in a safe and predictable manner.

We solve the delay sensitivity problem by copying the delay-sensitive pages to the hypervisor's inner buffer in advance, i.e., when the hypervisor is requested to start memory acquisition. The ASLR complication is addressed by inspecting the operating system's dynamic map of memory regions and obtaining the dynamic locations of the delay-sensitive pages. Section 4.2 contains a detailed description of ASLR as it is implemented in Windows 10 and our solution of the delay sensitivity problem.

The contribution of our work is:

1. We show how memory can be acquired on systems with multiple processors.
2. We present a solution to the delay sensitivity problem.
3. We explain how the locations of sensitive pages can be obtained dynamically on Windows 10. We believe that similar methods will be applicable to future versions of Windows.

2 RELATED WORK

Previous versions of the Windows operating system contained a special device, `\\Device\PhysicalMemory`, that mapped the entire physical memory. This device could be used to acquire the physical memory without any special tools. Unfortunately, because this method of memory acquisition relies on the operating system, a skilled attacker can disable or corrupt this feature (Carrier and Grand, 2004). Moreover, this method is not available in Windows 2003 SP1 and later versions of Windows (Microsoft Corporation, 2009).

Another method of memory acquisition is based on generic or dedicated hardware. Several previous works show how a generic FireWire card can be used to acquire memory remotely (Zhang et al., 2010). A dedicated PCI card, named Tribble, works in a similar manner (Carrier and Grand, 2004). The main advantage of a hardware solution is the ability of a PCI card to communicate with the memory controller directly, thus providing a reliable result even if the operating system itself was compromised. However, hardware solutions have three deficiencies:

1. They are expensive.
2. The produced memory image is not atomic.
3. These tools fail when Device Guard (Durve and Bouridane, 2017), a security feature introduced in Windows 10, is enabled.

Device Guard is a security feature that utilizes IOMMU (Ben-Yehuda et al., 2007; Zaidenberg, 2018) to prevent malicious access to memory from physical devices (Brendmo, 2017). When Device Guard is enabled, the operating system assigns each device a memory region that it is allowed to access. Any attempt to access memory outside this region is prevented by the DMA controller.

Recently, several hypervisor-based methods of memory acquisition have been proposed. HyperSleuth (Martignoni et al., 2010) is a driver with an embedded hypervisor. Its hypervisor is capable of performing atomic and *lazy* memory acquisition. The laziness is expressed in the ability of the hypervisor to continue the normal execution while the memory is acquired. ForenVisor (Qi et al., 2017) is a similar hypervisor with additional features that allow it to log keyboard strokes and hard-drive activity. Both hypervisors were tested on Windows XP SP3 with only one processor enabled.

We show how the idea of HyperSleuth and ForenVisor can be adapted to multi-processor systems executing Windows 10.

3 BACKGROUND

3.1 Hypervisors

The main component of the described system is a hypervisor, which utilizes the VMX instruction set extension. This section provides a short overview of this component. Section 4 contains a detailed description of the hypervisor's design There are two types of hypervisors: full hypervisors and thin hypervisors. Full hypervisors like Xen (Barham et al., 2003), VMware Workstation (VMware, 2018), and Oracle VirtualBox (Oracle, 2018) can execute several operating systems concurrently. The main goal of VMX was to provide software developers with means to construct efficient full hypervisors.

Thin hypervisors, in contrast, can execute only a single operating system. Their main purpose is to enrich the functionality of an operating system. The main benefit of a hypervisor over kernel modules (device drivers) is the hypervisor's ability to create an isolated environment, which is important in some cases. Thin hypervisors are used for operating system's integrity validation (Seshadri et al., 2007), remote attestation (Kiperberg et al., 2015; Kiperberg and Zaidenberg, 2013), malicious code execution prevention (Resh et al., 2017), in-memory secret protection (Resh and Zaidenberg, 2013), hard drive encryption (Shinagawa et al., 2009), and memory acquisition (Qi et al., 2017),

In general, because thin hypervisors are much smaller than full hypervisors, they are superior in their performance, security, and reliability. The hypervisor described in this paper is a thin hypervisor that is capable of acquiring a memory image of an executing system atomically. The hypervisor was written from scratch to achieve an optimal performance.

Similarly to an operating system, a hypervisor does not execute voluntarily but responds to events, e.g., execution of special instructions, generation of exceptions, access to memory locations, etc. The hypervisor can configure interception of (almost) each event. Interception of an event (a VM-exit) is similar to handling of an interrupt, i.e., a predefined function is executed by the processor. Another similarity with an operating system is the hypervisor's ability to configure the access rights to each memory page through a data structure, named EPT, which resembles the virtual page table. An attempt to write to a non-writable (according to EPT) page induces a VM-exit and allows the hypervisor to act.

3.2 Lazy Hypervisor-based Memory Acquisition

The solutions proposed by HyperSleuth and ForenVisor for memory acquisition are based on a thin hypervisor and can be summarized as follows. The hypervisor remains idle (or deactivated) until it receives a memory acquisition request. When the request is received, the hypervisor configures the EPT to make all memory pages non-writable. An attempt to write to a page P will trigger a VM-exit, thus allowing the hypervisor to react. The hypervisor reacts by copying P to an inner queue, and making P writable again. Future attempts to write to P will not trigger a VM-exit.

The queued pages are transmitted to a remote machine via a communication channel. This channel may be secure or not, depending on the security assumptions about the underlying environment. The size of the queue is dictated by the communication channel bandwidth and the volume of pages that are modified by the system. Obviously, if the communication channel allows sending more data than is available in the queue, then the hypervisor sends other non-writable pages and configures them to be writable. This process continues until all pages become writable.

3.3 Delay-sensitive Pages and ASLR

Generally, interrupt service routines react to interrupts in two steps: they register the occurrence of an interrupt and acknowledge the device that the interrupt was serviced. The acknowledgement must be received in a timely manner; therefore, the registration of an interrupt occurrence, which involves writing to a memory page, must not be intercepted by a hypervisor, i.e., these pages must remain writable. This issue was not addressed by the authors of HyperSleuth and ForenVisor. We assume that this problem did not occur on Windows XP SP3, which was tested in previous works.

Address space layout randomization, a security feature employed by modern operating system, e.g., Windows 10, complicates the delay sensitivity problem even more. When ASLR is enabled, the operating system splits its virtual address space into regions. Then, during the initialization of the operating system, each region is assigned a random virtual address. This behavior is useful against a wide range of attacks (Evtyushkin et al., 2016) because the location of potentially vulnerable modules is not known in advance. However, for the exact same reason, the location of the delay-sensitive pages is also unpredictable.

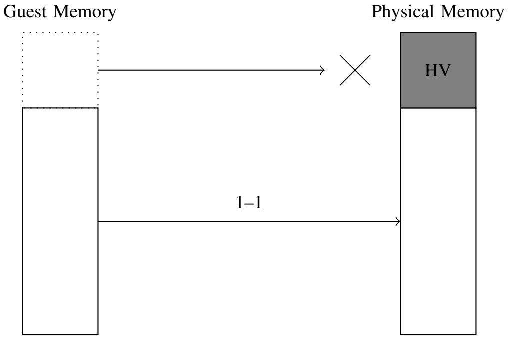

Figure 1: Mapping between the physical address space as observed by the operating system (left) and the actual physical address space. The mapping is an identity mapping with the exception of the hypervisor's pages, which are not mapped at all.

4 SYSTEM DESIGN

4.1 Initialization

Our hypervisor is implemented as a UEFI application (Unified EFI, Inc., 2006). The UEFI application loads before the operating system, allocates all the required memory, and initializes the hypervisor. After initialization, the UEFI application terminates, thus allowing the operating system boot loader to initialize the operating system. We note that while the application terminates, the hypervisor remains active.

In order to protect itself from a potentially malicious environment, the hypervisor configures the EPT such that any access to the code and the data of the hypervisor is prohibited. With this exception, the EPT is configured to be an identity mapping that allows full access to all the memory pages (Figure 1).

The hypervisor remains idle until an external event triggers its memory imaging functionality. The external event might be the reception of a network packet, insertion of a USB device, invocation of a system call, etc. In our prototype implementation, we used a special CPUID instruction, which we call `FREEZE`, as a trigger.

In response to `FREEZE`, the hypervisor performs two actions:

1. Locates and copies the delay-sensitive pages.
2. Requests all processors to configure the access rights of all memory pages to be non-writable.

When the configuration is complete, the hypervisor reacts to page modification attempts by making the page writable and copying it to an inner queue. The hypervisor exports the pages stored in the inner queue in response to another special CPUID instructions, which we call `DUMP`. If the queue is not full, then the hypervisor exports other non-writable pages and makes the exported pages writable.

Algorithm 1: Memory Acquisition.

```
file ← Open(...)
FREEZE()
while DUMP(addr, page) do
    Seek(file, addr)
    Write(file, page)
Close(file)
```

Algorithm 1 shows how `FREEZE` and `DUMP` can be used to acquire an atomic image of the memory. First, the algorithm opens a file that will contain the resulting memory image. Then, `FREEZE` is invoked, followed by a series of `DUMP`s. When the `DUMP` request returns *false*, the file is closed and the algorithm terminates.

4.2 Delay-sensitive Pages

Section 3.3 explains that certain pages must not be configured as non-writable. Moreover, due to ASLR, the hypervisor has to discover the location of these pages at run time based on the operating system data structures. This section presents the data structures of Windows 10 that can be used to locate the delay-sensitive pages.

Windows 10 defines a global variable `MiState` of type `MI_SYSTEM_INFORMATION`. The hypervisor can easily locate this variable as it has a constant offset from the system call service routine, whose address is stored in the LSTAR register (Table 1). The `MI_SYSTEM_INFORMATION` structure has a field named `Vs` of type `MI_VISIBLE_STATE`. Finally, the `MI_VISIBLE_STATE` structure has a field named `SystemVaRegions`, which is an array of 15 pairs. Each pair corresponds to a memory region whose address was chosen at random during the operating system's initialization. The first element of the pair is the random address and the second element is the region's size. A description of each memory region is given in Table 2. A more detailed discussion of the memory regions appears in (Russinovich et al., 2012). Our empirical study shows that that the following regions contain delay-sensitive pages:

1. MiVaProcessSpace
2. MiVaPagedPool
3. MiVaSpecialPoolPaged
4. MiVaSystemCache
5. MiVaSystemPtes
6. MiVaSessionGlobalSpace

Table 1: Windows ASLR-related Data Structures.

Offset	Field/Variable Name	Type
X	System Call Service Routine	*Code*
...	...	...
+0xFB100	MiState	MI_SYSTEM_INFORMATION
+0x1440	Vs	MI_VISIBLE_STATE
+0x0B50	SystemVaRegions	MI_SYSTEM_VA_ASSIGNMENT[14]
+0x0000	[0]	MI_SYSTEM_VA_ASSIGNMENT
+0x0000	BaseAddress	uint64_t
+0x0008	NumberOfBytes	uint64_t

Table 2: Memory Regions.

Index	Name
0	MiVaUnused
1	MiVaSessionSpace
2	MiVaProcessSpace
3	MiVaBootLoaded
4	MiVaPfnDatabase
5	MiVaNonPagedPool
6	MiVaPagedPool
7	MiVaSpecialPoolPaged
8	MiVaSystemCache
9	MiVaSystemPtes
10	MiVaHal
11	MiVaSessionGlobalSpace
12	MiVaDriverImages
13	MiVaSystemPtesLarge

Therefore, the hypervisor never makes these regions non-writable.

4.3 Multiprocessing

The hypervisor responds to `FREEZE`, a memory acquisition request, by copying the delay-sensitive pages to an inner queue and configuring all other pages to be non-writable. However, when multiple processors are active, the access rights configuration must be performed atomically on all processors.

Operating systems usually use inter-processor interrupts (IPIs) (Intel Corporation, 2018) for synchronization between processors. It seems tempting to use IPIs also in the hypervisor, i.e., the processor that received `FREEZE` can send IPIs to other processors, thus requesting them to configure the access rights appropriately. Unfortunately, this method requires the hypervisor to replace the operating system's interrupt-descriptors table (IDT) with the hypervisor's IDT. This approach has two deficiencies:

1. Kernel Patch Protection (KPP) (Field, 2006), a security feature introduced by Microsoft in Windows 2003, performs a periodic validation of critical kernel structures in order to prevent their illegal modification. Therefore, replacing the IDT requires also intercepting KPP's validation attempts, which can degrade the overall system performance.
2. Intel processors assign priorities to interrupt vectors. Interrupts of lower priority are blocked while an interrupt of a higher priority is delivered. Therefore, the hypervisor cannot guarantee that a sent IPI will be handled within a predefined time. Suspending the operating system for long periods can cause the operating system's watchdog timer to trigger a stop error (BSoD).

We present a different method to solve the inter-processor synchronization problem that is based on a documented functionality of the operating system itself. The `KeIpiGenericCall` function (Microsoft Corporation, 2018) receives a callback function as a parameter and executes it on all the active processors simultaneously. We propose to use the `KeIpiGenericCall` function to configure the access rights simultaneously on all the processors.

Because it is impossible to call an operating system function from within the context of the hypervisor, the hypervisor calls the `KeIpiGenericCall` function from the context of the (guest) operating system. In order to achieve this, the hypervisor performs several preparations and then resumes the execution of the operating system.

Algorithm 2 presents three functions that together perform simultaneous access rights configuration on all the active processors. The first function, HANDLECPUID, is part of the hypervisor. This function is called whenever the operating system invokes a special `CPUID` instruction. Two other functions, GUESTENTRY and CALLBACK, are mapped by the hypervisor to a non-occupied region of the operating system's memory.

Algorithm 1 begins with a special `CPUID` instruction, called `FREEZE`. This instruction is handled by lines 2–5 in Algorithm 2: the hypervisor maps GUESTENTRY and CALLBACK, saves the current registers' values and sets the instruction pointer to the address of GuestEntry. The GuestEntry function calls the operating system's KeIpiGenericCall, which will execute Callback on all the active processors. The Callback function performs another special CPUID instruction, called CONFIGURE, which causes the hypervisor to configure the access rights of all (but the delay-sensitive) memory pages on all the processors. This is handled by lines 6–7 of the algorithm, where we omitted the configuration procedure itself. After the termination of the Callback function, the control returns to the GuestEntry function, which executes a special CPUID instruction, named RESUME_OS. In response, the hypervisor restores the registers' values, which were previously saved in line 4. The operation continues from the instruction following FREEZE, which triggered this sequence of events.

Algorithm 2: Simultaneous access rights configuration on all the active processors.

```
 1: function HandleCpuid(reason)
 2:     if reason=FREEZE then
 3:         Map GuestEntry and Callback
 4:         Save registers
 5:         RIP ← GuestEntry
 6:     else if reason=CONFIGURE then
 7:         ...
 8:     else if reason=RESUME_OS then
 9:         Restore registers
10:     else if reason=DUMP then
11:         ...
12:     ...
13: function GuestEntry
14:     KeIpiGenericCall(Callback)
15:     CPUID(RESUME_OS)
16: function Callback
17:     CPUID(CONFIGURE)
```

5 EVALUATION

This section evaluates the performance of the HV and its memory usage. First, we demonstrate the overall performance impact of the HV. Next, we analyze the memory usage of the HV. Finally, we evaluate the performance of the memory acquisition process.

All the experiments were performed in the following environment:

- CPU: Intel Core i5-6500 CPU 3.20GHz (4 physical cores)
- RAM: 16.00 GB
- OS: Windows 10 Pro x86-64 Version 1803 (OS Build 17134.407)

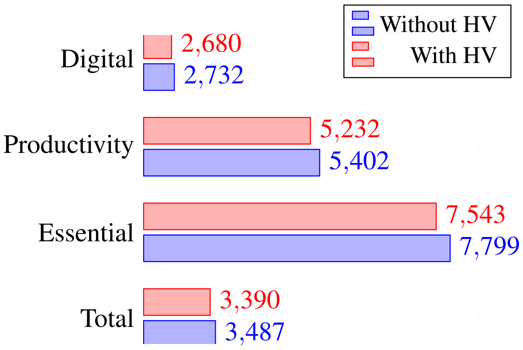

Figure 2: Scores (larger is better) reported by PCMark in four categories: Digital Content Creation, Productivity, Essential, and Total.

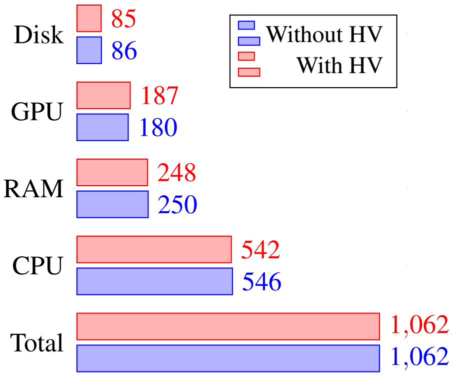

Figure 3: Scores (larger is better) reported by Novabench in five categories: Disk, GPU, RAM, CPU, and Total.

- C/C++ Compiler: Microsoft C/C++ Optimizing Compiler Version 19.00.23026 for x86

5.1 Hypervisor Performance Impact

We start by demonstrating the performance impact of the hypervisor on the operating system. We picked two benchmarking tools for Windows:

1. PCMark 10 – Basic Edition. Version Info: PCMark 10 GUI – 1.0.1457 64 , SystemInfo – 5.4.642, PCMark 10 System 1.0.1457,
2. Novabench. Version Info: 4.0.3 – November 2017.

Each tool performs several tests and displays a score for each test. We invoked each tool twice: with and without the hypervisor. The results of PCMark, and Novabench are depicted in Figures 2–3, respectively. We can see that the performance penalty of the hypervisor is approximately 5% on average.

Table 3: Memory Regions' Sizes.

Index	Name	Size (MB)
0	MiVaUnused	6
1	MiVaSessionSpace	100
2	MiVaProcessSpace	0
3	MiVaBootLoaded	0
4	MiVaPfnDatabase	0
5	MiVaNonPagedPool	6
6	MiVaPagedPool	0
7	MiVaSpecialPoolPaged	5
8	MiVaSystemCache	52
9	MiVaSystemPtes	0
10	MiVaHal	0
11	MiVaSessionGlobalSpace	0
12	MiVaDriverImages	8

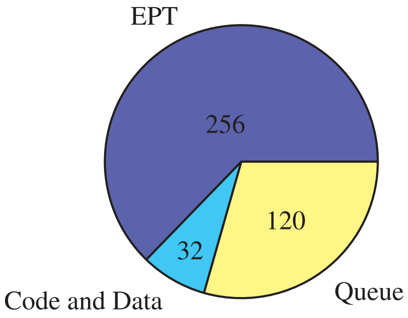

Figure 4: Hypervisor's Memory Usage [MB].

5.2 Memory Usage

The memory used by the hypervisor can be divided into three main parts:

1. the code and the data structures of the hypervisor,
2. the EPT tables used to configure the access rights to the memory pages, and
3. the queue used to accumulate the modified pages.

Figure 4 presents the memory usage of the hypervisor including its division.

The size of the queue is mainly dictated by the number of delay-sensitive pages. Table 3 presents the typical size of each memory region. Pages belonging to the following regions are copied by the hypervisor:

1. MiVaProcessSpace
2. MiVaPagedPool
3. MiVaSpecialPoolPaged
4. MiVaSystemCache
5. MiVaSystemPtes
6. MiVaSessionGlobalSpace

Their total size is $\approx$ 60MB. The size of the queue should be slightly larger than the total size of the delay-sensitive pages because regular pages can be modified by the operating system before the content of the queue is exported. Our empirical study shows that it is sufficient to enlarge the queue by 60MB.

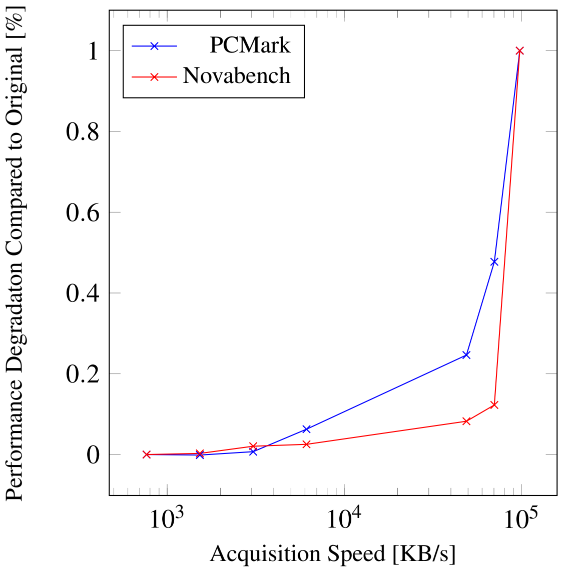

Figure 5: Performance degradation due to memory acquisition.

5.3 Memory Acquisition Performance

In this section we study the correlation between the speed of memory acquisition and the overall system performance. Figure 5 shows the results. The horizontal axis represents the memory acquisition speed. The maximal speed we could achieve was 97920KB/s. At this speed, the system became unresponsive and the benchmarking tools failed. The vertical axis represents the performance degradation (in percent) measured by PCMark and Novabench. More precisely, denote by $t_i(x)$ the *Total* result of benchmark $i = 1,2$ (for PCMark and Novabench, respectively) with acquisition speed of x; then, the performance degradation $d_i(x)$ is given by $d_i(x) = 1 - \frac{t_i(x)}{t_i(0)}$.

6 CONCLUSIONS

The method presented in this work should be seen as an incremental improvement over previously described methods, which have similar purpose and design. We describe two improvements over the currently available methods:

1. Our hypervisor supports multiple processors by utilizing an operating system's function for processor synchronization.

2. Our hypervisor supports modern operating systems, e.g., Windows 10, by locating and copying the delay-sensitive pages.

Section 5 presents the memory usage of the hypervisor. We believe that this memory usage can be improved by reducing the number of pages that the hypervisor considers to be delay-sensitive.

REFERENCES

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, volume 37, pages 164–177. ACM.

Ben-Yehuda, M., Xenidis, J., Ostrowski, M., Rister, K., Bruemmer, A., and Van Doorn, L. (2007). The price of safety: Evaluating iommu performance. In *The Ottawa Linux Symposium*, pages 9–20.

Brendmo, H. K. (2017). Live forensics on the windows 10 secure kernel. Master's thesis, NTNU.

Carrier, B. D. and Grand, J. (2004). A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation*, 1(1):50–60.

Cohen, M. (2014). Rekall memory forensics framework. *DFIR Prague*.

Durve, R. and Bouridane, A. (2017). Windows 10 security hardening using device guard whitelisting and applocker blacklisting. In *Emerging Security Technologies (EST), 2017 Seventh International Conference on*, pages 56–61. IEEE.

Evtyushkin, D., Ponomarev, D., and Abu-Ghazaleh, N. (2016). Jump over aslr: Attacking branch predictors to bypass aslr. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 40. IEEE Press.

Field, S. (2006). An introduction to kernel patch protection. http://blogs.msdn.com/b/windowsvistasecurity/archive/2006/08/11/695993.aspx.

Intel Corporation (2018). *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation.

Kiperberg, M., Resh, A., and Zaidenberg, N. J. (2015). Remote attestation of software and execution-environment in modern machines. In *Cyber Security and Cloud Computing (CSCloud), 2015 IEEE 2nd International Conference on*, pages 335–341. IEEE.

Kiperberg, M. and Zaidenberg, N. (2013). Efficient remote authentication. In *Proceedings of the 12th European Conference on Information Warfare and Security: ECIW 2013*, page 144. Academic Conferences Limited.

Martignoni, L., Fattori, A., Paleari, R., and Cavallaro, L. (2010). Live and trustworthy forensic analysis of commodity production systems. In *International Workshop on Recent Advances in Intrusion Detection*, pages 297–316. Springer.

Microsoft Corporation (2009). Device\PhysicalMemory Object. https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc787565(v=ws.10). [Online; accessed 02-Nov-2018].

Microsoft Corporation (2018). KeIpiGenericCall function. https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/nf-wdm-keipigenericcall.

Oracle (2018). VirtualBox. https://www.virtualbox.org/.

Qi, Z., Xiang, C., Ma, R., Li, J., Guan, H., and Wei, D. S. (2017). Forenvisor: A tool for acquiring and preserving reliable data in cloud live forensics. *IEEE Transactions on Cloud Computing*, 5(3):443–456.

Reina, A., Fattori, A., Pagani, F., Cavallaro, L., and Bruschi, D. (2012). When hardware meets software: a bulletproof solution to forensic memory acquisition. In *Proceedings of the 28th annual computer security applications conference*, pages 79–88. ACM.

Resh, A., Kiperberg, M., Leon, R., and Zaidenberg, N. J. (2017). Preventing execution of unauthorized native-code software. *International Journal of Digital Content Technology and its Applications*, 11.

Resh, A. and Zaidenberg, N. (2013). Can keys be hidden inside the cpu on modern windows host. In *Proceedings of the 12th European Conference on Information Warfare and Security: ECIW 2013*, page 231. Academic Conferences Limited.

Russinovich, M. E., Solomon, D. A., and Ionescu, A. (2012). *Windows internals*. Pearson Education.

Seshadri, A., Luk, M., Qu, N., and Perrig, A. (2007). Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 335–350. ACM.

Shinagawa, T., Eiraku, H., Tanimoto, K., Omote, K., Hasegawa, S., Horie, T., Hirano, M., Kourai, K., Oyama, Y., Kawai, E., Kono, K., Chiba, S., Shinjo, Y., and Kato, K. (2009). Bitvisor: A thin hypervisor for enforcing i/o device security. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, pages 121–130, New York, NY, USA. ACM.

Unified EFI, Inc. (2006). Unified Extensible Firmware Interface Specification, Version 2.6.

VMware (2018). VMware Workstation Pro. https://www.vmware.com/il/products/workstation-pro.html.

Zaidenberg, N. J. (2018). Hardware rooted security in industry 4.0 systems. In Dimitrov, K., editor, *Cyber defence in Industry 4.0 and Related Logistic and IT Infrastructures*, chapter 10, pages 135–151. IOS Press.

Zhang, L., Wang, L., Zhang, R., Zhang, S., and Zhou, Y. (2010). Live memory acquisition through firewire. In *International Conference on Forensics in Telecommunications, Information, and Multimedia*, pages 159–167. Springer.