Hypervisor Memory acquisition for ARM

Raz Ben Yehuda, Erez Shlingbaum, Shaked Tayouri, Yuval Gershfeld, Nezer Zaidenberg

Abstract

Cyber forensics use memory acquisition in advanced forensics and malware analysis. We propose a hypervisor based memory acquisition tool. Our implementation extends the volatility memory forensics framework by reducing the processor's consumption, solves the incoherency problem in the memory snapshots and mitigates the pressure of the acquisition on the network and the disk. We provide benchmarks and evaluation.

1 Introduction

A rootkit is a malware that hides itself along with the malicious payload that it carries[8]. Rootkit research is a cat a mouse game in all computing platforms.

Researchers develop better forensics to detect rootkits while others develop state-of-the-art rootkits. Our research improves a volatile memory forensics. We describe a hypervisor-based forensics software that detects stealthy rootkits under Linux. We enhanced LiME's [29] memory acquisition tool for Volatility [12] memory forensics software.

We describe our the design and implementation of an online memory forensics system. We also performed a performance analysis of memory acquisition performance figures on the online system. This paper outline is as follows:

We start by describing the problem's Background. We then detail our Contribution and present an Evaluation. We end with Related work and Conclusions.

2 Background

2.1 Rootkits and stealth malware

Two of the most famous rootkits of recent years were stuxnet [16] and Sony BMG [19] rootkit. Stuxnet purpose was to attack Iran's nuclear enrichment program. Countries and elite intelligence agencies developed Stuxnet for espionage and Sabotage purposes. Sony BMG rootkit designed to install and hide DRM software on end-user machines [13, 4]. These two examples demonstrate the rootkits are no longer "hacker tools" but rather tools employed by countries and top industrial companies for national and business purposes.

Despite having completely different authors and purposes, both software contained a similar concept of masking its existence by hiding the malware files and the running processes. Live memory acquisition is a tool used by forensics researchers to reverse engineer the malware. Forensics researchers attempt to identify the authors, their goals and any weakness in the malware itself.

Volatility [12, 9] is an open-source (GPLv2) framework for analysing memory [5]. It is a forensics toolkit used to analyze memory snapshots. Thus Volatility is often used to detect such hidden malware. [20]

2.2 Forensics and Volatility

Volatility is the state of the art RAM snapshot software. It is available in Windows, Linux, Mac, and Android, 32bit and partially 64bit. Volatility is based on Python [28]. which makes development in Volatility easy for most analysts. Also, Python is available on many operating systems, including Android phones. Volatility availability on Android allows for local memory capture and analysis, saving the need to transmit gigabytes of a RAM snapshot over the network. Volatility API is extensible, and forensics researchers can add plugins easily. Volatility's developers use a reverse engineering approach to understand the acquired memory. Thus, Volatility provides capabilities and information that are not usually possible through standard tools. For example, examining undocumented data structures in windows OS. Volatility supports various formats: crash dumps, hibernation files, VMware's vmem, VMware's saved state and

suspended files (.vmss/.vmsn), VirtualBox core dumps, LiME (Linux Memory Extractor), expert witness (EWF), and direct physical memory over Firewire. Volatility is considered fast compared to other forensics tools. It analyses the entire memory image file in a few seconds.

2.3 LiME

LiME (Linux Memory Extractor) is a Linux kernel module that performs acquisition of volatile memory on Linux distributions and Linux kernel-based devices [11], such as Android. Since LiME is a Linux kernel module, it does not require any user-space tools to perform memory captures. Therefore, LiME memory captures are considered sounder than other memory acquisition tools. Also, since LiME is a pure kernel module, it is easy to use it on Android devices and embedded Linux devices in general.

2.4 ARM permission model

ARM has a unique approach to security and privilege levels [22] that is crucial to the implementation of our microvisor. In ARMv7, ARM introduced the concept of secured and non-secured worlds through the implementation of TrustZone [30]. ARM architecture includes four exceptions (permission) levels as follows.
Exception Level 0 (EL0) Refers to the user-space. Exception Level 0 is analogous to "ring 3" on the x86 platform.
Exception Level 1 (EL1) Refers to the operating system. Exception Level 1 is analogous to "ring 0" on the x86 platform.
Exception Level 2 (EL2) Refers to the hypervisor (HYP mode). Exception Level 2 is analogous to "ring -1" or "real mode" on the x86 platform.
Exception Level 3 (EL3) Refers to the TrustZone as a special security mode that can monitor the ARM processor and may run a real-time security OS. There are no direct analogous modes, but related concepts in x86 are Intel's ME or SMM.

Each exception level provides its own set of special-purpose registers and can access these registers of the lower levels, but not higher levels. The general-purpose registers are shared. Thus, moving to a different exception level on the ARM architecture does not require the expensive context switch that is associated with the x86 architecture.
In the context of the paper, we use EL2 microvisor to extend LiME.

3 Contribution

3.1 High level design

We use Volatility [26] and LiME to analyse the memory to detect an irregular state. Our contribution focuses on reducing CPU consumption and heat when using LiME. Our memory acquisition microvisor also provides consistent memory images. Also, ported parts of Volatility to support 64bit ARM Linux kernels.

3.2 ARM systems

Volatility currently supports ARMv7 systems. One key difference between ARMv7 and ARMv8 regarding memory acquisition is that ARMv8 usually runs 64bit kernels while ARMv7 runs 32bit. Our microvisor environment does not support ARMv7; therefore, we ported Volatility to ARM64 bit kernels partially and contributed our modifications to the Volatility community. This approach is important because most Phones and Android devices today use ARMv8-a and a 64bit Linux kernel.

3.3 Microvised LiME

Our contribution concentrates in LiME. LiME creates a reflection of the computer's RAM by accessing each physical page and writing it to the disk or the network (Algorithm 1). LiME scans the RAM sequentially and allocates an auxiliary page for each page and copies the page's contents to it. The auxiliary page is transmitted (or written to disk) and then released.

Algorithm 1: LiME Main Transmission

```
while not end of RAM do
    Map the current physical page the to kernel
      memory
    Allocate an auxiliary page
    Copy Physical Page to the auxiliary page
    Transmit the auxiliary Page
    Free auxiliary page
    Unmap the Physical Page
end
```

This paper contribution focuses on solving two disadvantages of LiME:

- The snapshot image in-coherency
- The processor's high utilisation while the acquisition takes place.
- LiME is unable to predetermine the chances of an in-coherent memory snapshot.

3.4 Memory In-coherency

The main memory is likely to change during the acquisition. For example, Figure 1 demonstrates a browser launch during LiME memory acquisition, and a few seconds after the browser immediately exits. The operation changes the operating system's processes table after LiME already transmitted it. Therefore, starting and stopping Firefox creates memory in-coherency. Memory in-coherence is a result of processes that exist in memory but not on the process table, pages that belong to a process that does not exist in the table, etc.

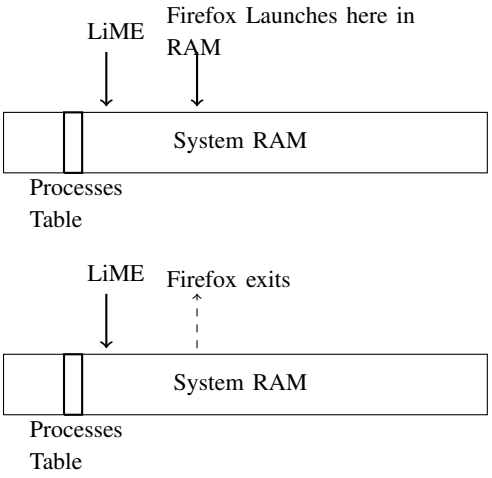

Figure 1: LiME transmission

Andrew et al. [9] describe other in-coherency reasons, such as multiples cores, the increasing amount of RAM and the kernel heuristics, and point the increasing amount of RAM as the reason for page smearing. In this paper context, [9] mention two techniques to deal with page smearing:

- **Leveraging virtual machine hardware extensions** Use various virtualization techniques, such as blue pill, to acquire the memory.
- **Smear-aware acquisition tools.** Provide the acquisition tool awareness to changes in page tables.

Our technology utilizes virtualization to acquire memory, but without freezing the operating system (hibernation).

To emphasise how versatile the RAM (and page smearing) while LiME transmits, we recorded the first 500 page faults positions while LiME works. Table 1 presents the distance (in pages) between two consequent pages. The total number of pages is 242000. We performed the test above while the system was idle.

Avg	Min	Max	Std dev
426	-236990	236921	44837

Table 1: Page faults distribution in pages

The pages' positions distribution is vast. Therefore, we expected to find inconsistencies in the memory image snapshot.

Thus, this paper offers a technique to solve the inconsistency. We wrap the GPOS (General Purpose Operating System) by a minimal virtual machine (VM), and during the acquisition, we record and copy the faulting page content before it is transmitted. In virtual machines, when a guest accesses a page, the Memory Management Unit (MMU) traps it to a secondary page table managed by the hosting machine. This mechanism is referred to as a stage 2 fault. In ARM, the second (stage 2) memory table is called the Intermediate Physical Addresses (IPA) [23].

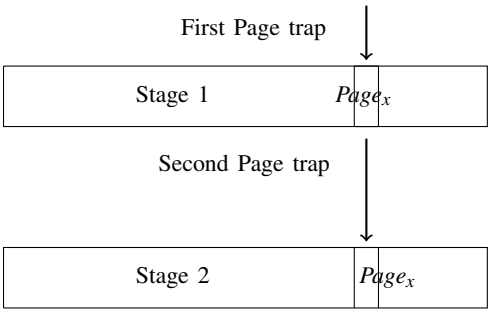

Figure 2: Virtual Machine dual stage memory trap

Thus, to achieve coherency, we set all stage 2 tables to read-only before the memory acquisition starts. This way any write access to any page is trapped to the microvisor. We then execute LiME to acquire the computer's RAM; at this point, any page that traps to the microvisor is copied to a **pages pool** and the real page permissions are set to read-write. Therefore each page can trap to the microvisor at most once.

In addition to our microvisor, our solution includes a modification to the LiME driver. LiME, using our technique, linearly scans the memory, and in each cycle it checks the pages pool to verify that the current page did not already fault. If the page is resident in the pool, LiME transmits the copied page and releases it back to the microvisor's pool for re-use. Algorithm 2 describes LiME's new implementation while using our microvisor. We refer in this paper to this technique as a **linear acquisition**.

Algorithm 2: Linear Acquisition: Transmission with a microvisor

```
while not end of RAM do
    Allocate an auxiliary page
    if the physical page has an old version in the
     pool then
        Remove the page from pages pool
        Copy the page from the pool to the auxiliary
         page
        Place back the page to the pool
        else
            Map Physical Page
            Copy physical page to the auxiliary page
            Unmap Physical Page
        end
    end
    Transmit auxiliary Page
    Free auxiliary page
end
```

3.5 CPU consumption

Linear acquisition has a flaw. As we show later in the evaluation section, the performance cost of linear acquisition is very high. To reduce the processor consumption, we modified LiME's main transmission routine to be less CPU intense. In this approach, LiME does not scan the memory linearly. Instead LiME removes pages from the pool as long as there are pages in it. If there are no pages, it linearly sends pages as before, but it sleeps for a few milliseconds in each transmission cycle. We refer to this solution as a **non linear acquisition**.

3.6 Refraining from incoherent images

As we've shown, LiME produces in-coherent memory dumps. Our microvised LiME solution offers a technique that produces cohesive memory images; However, in cases where there are too many page faults, it is not possible to create a coherent memory image, because the pages pool is overloaded. In this case, it is easy to disable the acquisition and starts again later.

3.7 Microvisor memory model TEE

Here we explain the trusted execution environment(TEE) model. ARMv8-a hypervisor (EL2) memory table is not accessible to lower exception levels; meaning, it is not possible for code running in EL2 to access another exception level memory without premature mappings (Figure 3).

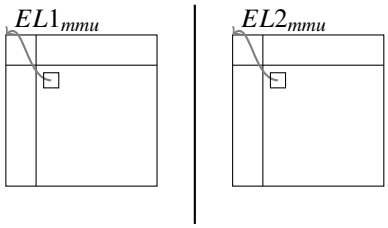

Figure 3: Memory Table access

For instance, to access memory in EL1 from EL2, EL1's pages must be pre-mapped (Figure 4) to EL2.

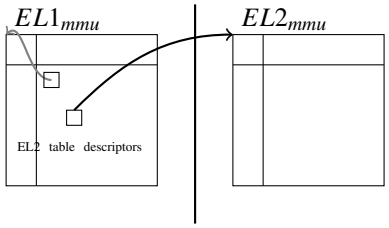

Figure 4: Memory Table access for hyplets

However, during the acquisition, each page accessed in EL1 or in EL0 traps to EL2 (stage 2), and therefore, must be copied to a page in the microvisor pages pool, while in the EL2 trap. So it is essential to map this page to microvisor as done in KVM for ARM [10].
Our hypervisor page trap needs to access both virtual userspace and virtual kernel space addresses. Unfortunately, ARMv8-a hypervisor MMU is capable of handling only user-space addresses (the upper 16 bits are zeroes) and does not support kernel virtual addresses (upper 16 bits are ones) at all. So instead of handling virtual addresses, we decided to handle physical addresses. We rely on the following ARM property. When a page traps to EL2, in addition to the virtual address of the faulting page, the physical address of the page is also reported in a special register.

Therefore, we map the entire physical memory to the hypervisor (Figure 5) as-is, meaning, we map each address without any offset. This way, the code, and some data are mapped in a certain part of the address space, represented by a very high number; while the entire System's RAM resides in address starting from address 0 (the lowest physical address). To summarise, we have two distinct mappings techniques:

- General Mappings

  hypervisor map(kernel addr) = address + offset

  These mappings are used to execute code and access general management data from both the kernel and the hypervisor.

- Physical mappings

  hypervisor map2(physical address) = physical address

  These mappings are used only in the page trap function of the microvisor. The trap copies the data to be used later by LiME.

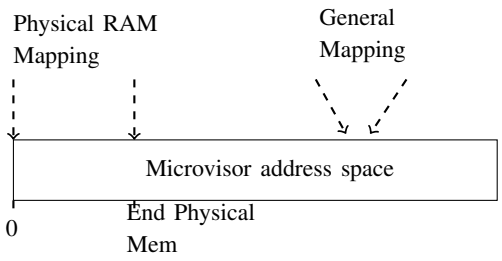

Figure 5: Microvisded LiME's Memory layout

All the pages in the General Mapping are mapped in the physical mapping as well. We note that some of these pages owned to the microvisor and never accessed by EL1/EL0 and therefore never trap to EL2 (Our microvisor traps from EL1 and EL0).

3.8 Acquisition Resources Consideration

This section discusses the memory resources required for the acquisition. Figure 6 demonstrates the fluctuations in the number of pages accessed in each processor in our hardware. It is evident that there is a burst in the number of pages as LiME starts and when the acquisition ends.

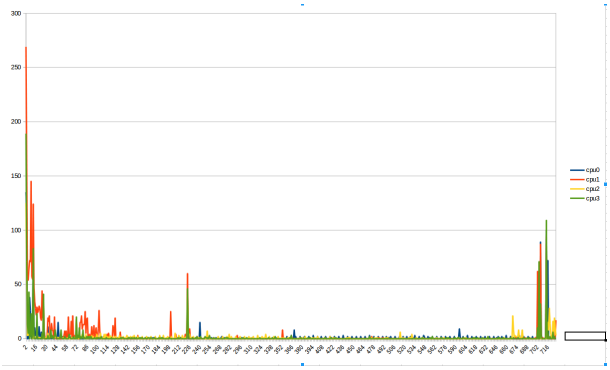

Figure 6: #pages accessed to the microvisor. X axis is #samples

In our hardware, nearly 600 pages are accessed as the acquisition starts. So, we expect our pool to be in this magnitude. Let us examine the overhead of 1000 pages assigned to the pool in the hardware of 242000 pages (PI3 - our evaluation hardware, has 940 MB of RAM). Thus the pages pool requires less than half per cent of the total RAM:

$$\frac{1000}{242000} \sim 0.4\%$$

To summarise this section, our acquisition algorithm incurs a temporary negligible overhead of RAM.

4 Evaluation

Our evaluation measures the IPA overhead, offers a performance comparison between the current implementation of LiME to the microvised implementation and demonstrates how RAM in-coherency is solved by our technology.

In the following evaluation, we repeat each test 10 times; We usually provide results in Idle mode and then Busy mode. Idle mode means that we did not apply any artificial stress on the operating system. A busy mode is when we applied the stress tool *stress* [1], while LiME was executing, as follows:

```
stress --VM bytes 128M --timeout 80s --VM 4
```

To measure memory speed, we used RAMspeed [**?**].

4.1 IPA

In Figure 3 we attempted to simulate more closely real-world computing load through the use of RAMspeed. A, B and C are locations in the memory. M in SCALE is a constant.

SCALE	A = m*B
ADD	A + B = C
COPY	A=B

Table 2: IPA Tests explained

Test	COPY	SCALE	ADD
Single Stage	2.76	1.52	2.60
Dual Stage	2.74	1.52	2.53

Table 3: Real World Load GB/s

There is no difference when using a two-stage translation to a single-stage translation. We have shown that IPA does not influence memory access performance.

4.2 Image in-coherency

We first demonstrate the in-coherency of RAM dumps when using LiME without a microvisor. We also show that the microvised LiME provides coherent RAM dumps. We perform the following test. We acquire memory while issuing the executing the "sleep 1" command 50 times sequentially for 1 second. A consistent snapshot should have a single instance of the sleep process in the process table. Figures 8 and Figure 7 are snippets of Volatility memory analysis of the processes tables. Figure 7 presents the processes tables when the non-microvised LiME is used, while Figure 8 is when using the microvised LiME. In Figure 7 we can see 27 instances of "sleep" process while in Figure 8 there is a single instance of "sleep". This means that LiME recorded the memory while the process table was changing. In Figure 7 we can see that some of the "sleep" process ids are successive, which means that no other processes were launched in at least 1 second. This strengthens the claim that even in an Idle system, in-coherency is possible.

```
Offset   Name       Pid PPid
------------------ ------------
0x2fc58000 sleep    641  -
0x308f8000 sleep    600  -
0x308f9d00 sleep    601  -
0x308fba00 sleep    602  -
0x308fd700 sleep    603  -
0x30918000 insmod   700  -
0x30919d00 sudo     708  -
0x30948000 swapoff  738  -
0x30978000 sleep    637  -
0x30979d00 sleep    638  -
0x3097ba00 sleep    639  -
0x3097d700 sleep    640  -
0x30980000 sleep    592  -
0x30981d00 sleep    593  -
0x30983a00 sleep    594  -
0x30985700 sleep    595  -
0x309c8000 sleep    596  -
0x309c9d00 sleep    597  -
0x309cba00 sleep    598  -
0x309cd700 sleep    599  -
0x30a10000 sleep    604  -
0x30a11d00 sleep    605  -
0x30a13a00 sleep    606  -
0x30a15700 sleep    607  -
0x30a38000 wpasupplicant 759  -
0x30a39d00 wireless-tools 753 -
0x30ac0000 swapon   735  -
0x30ac1d00 grep     736  -
0x30ac3a00 cut      737  -
0x30b98000 start-stop-daem 731 -
0x30ba8000 sleep    629  -
0x30ba9d00 sleep    634  -
0x30baba00 sleep    635  -
0x30bad700 sleep    636 -
....
```

Figure 7: Non-Microvised Lime

```
Offset   Name       Pid     PPid
------------------ -----    -------
0x2fc58000 sleep    641      -
0x3000fa00          318      -
0x3001e700 btuart   319      -
0x30035700 modprobe 508      -
0x30051000 ksoftirqd/0  9    -
x30053700 ksoftirqd/1  17    -
0x3007f000 ksoftirqd/2  22   -
0x304e0000 bash 527  -  0    -
0x304e1d00 top  530 -   0    -
..
```

Figure 8: Microvised LiME

Now, we want to have some measures for the possible in-coherency while LiME executes. So, we examine the number of EL2 page faults (Table 4) in Busy mode and Idle mode.

	Avg	Max	Min	Std dev
Idle Mode	130	129	132	1.17
Busy mode	128212	135891	122771	3493

Table 4: Total number of page faults during LiME

Table 4 was produced by the average and other statistical measures of 10 runs. We note again that for a pool of 1000 pages the in-coherency should be less than 0.4%. In Idle mode:

$$\frac{130}{242000} \sim 0.00005$$

but in Busy mode:

$$\frac{128212}{242000} \sim 0.52$$

The amount of in-coherency in the extreme case is approximately 50%. So our model can be $1000/130 \sim 7.5$ busier than the Idle mode presented here, and $1000/128212 \sim 0.0008$ times smaller than Busy mode. It is more efficient to run LiME when the system is Idle.

4.3 Microvised LiME vs LiME

We compare the performance of microvisied LiME to the non-microvised LiME. We measure the duration in seconds, in Idle mode and Busy mode, and the processor consumption in Idle mode. In the tests presented in tables 5 and 6 we used linear acquisition.

	Avg	Min	Max	Std dev
No microvisor	74	72	76	1.4
With a microvisor	80	74	96	7.68

Table 5: Duration of the memory dump, Idle mode

LiME	Avg	Min	Max	Std dev
No microvisor	84	83	85	0.7
With a microvisor	83	81	85	0.9

Table 6: Duration of the memory dump, Busy Mode

From tables 5 and 6 it is evident there is an overhead of 8% when using the microvised LiME. The reason is that for each page faulting to the microvisor, we scan the pool.
Lastly, we measure processor utilisation with and without a microvisor.

LiME	Avg	Min	Max	Std dev
no microvisor	52	50	53	1
With a microvisor	48.8	39	52	4.5

Table 7: Processor utilisation in Idle mode

Table 7 presents the overhead of the processor utilisation when LiME acquires memory. There is little difference between, so it is evident that even the linear version of microvised LiME does not incur any significant performance risks.

4.4 Non-Linear acquisition

Here we demonstrate the processor's consumption and the time it takes to perform a nonlinear acquisition. The variable we change in tables 8 and 9 is the delay duration in each cycle. We provide measures of the processor's consumption and the duration of the entire acquisition.

Test Conf	Duration
10 ms	4860 secs
20 ms	6600 secs
50 ms	14520 secs

Table 8: Duration of non linear acquisition

Test Conf	Avg Processor consumption
10 ms	17 %
20 ms	12%
50 ms	5%

Table 9: Processor utilization in Idle mode - Non Linear

Tables 8 and 9 prove that it is possible to perform an acquisition with very little processor consumption. The trade-off is a considerably long acquisition duration, in some cases over an hour. An additional benefit of this technique is that it has a low I/O load. The method does not congest the disk, or in the network case, mild network traffic. Thus it is possible to run the acquisition over a cellular medium.

5 Related work

In recent years it was proposed to use hypervisors for many security purposes such as machine introspection and debugging [32]. The introspection of virtual machines by the hypervisor was researched heavily. Libvmi [21] is a library that provides such introspection services under KVM. It provides VM based snapshots and has an integrated volatility plugin. It was also suggested to use Lguest[27] or Xen[6] for detection of kernel bugs[14], profiling[18], Hypertracing [7], security issues [31], and access the guest's memory through a thin hypervisor for remote attestation as suggested by Kiperberg et al. [15]. Forenvisor [25] uses the hypervisor to grab and store forensics data on the cloud for later inspection. Kiperberg et al. [15] provided a system for atomic memory acquisition and guaranteed atomic access.
Andrew et al. [9] discuss page swapping and demand paging as another obstacle to complete the acquisition of memory. Our technique does not acquire swap space or fetches pages of incomplete processes. At this stage, it is not clear whether the Volatility framework is capable to incorporate swap space and on-demand pages.
In Microsoft Windows operating systems, projects, like Powershell Empire [3], exploit Windows PowerShell. Powershell Empire does not execute the PowerShell executable file from the file-system but runs directly from the memory. Empire provides keylogging, credential theft, and more. As a result, memory forensics tools fail to detect the execution of PowerShell and forced to rely on pattern matching and search for side-effects of PowerShell post-execution. Another challenge is the Windows .NET framework. .NET runtime is embedded in the process's address space and, therefore, can execute an injected malware [24, 9]. At the moment there is no available technology that can detect this malware. Additionally, .NET supports function overriding; a technique that malware can manipulate by replacing callbacks with malicious functions.
In Android, a large effort is in the analysis of the Dalvik engine. Researchers created forensics tools [17] for Dalvik. Unfortunately, Dalvik was replaced by ART (Android Runtime). Unlike Dalvik which is based on JIT (Just in Time compilation), ART [2] produces ELF binaries. To the day of writing, there is no published

forensics technology for ART.
There have been multiple suggestions for memory inspection and acquisition through dedicated firmware. [33] et al. describe such memory acquisition through RDMA. In this paper, we assume that such hardware is not available.

6 Conclusions

We have shown that it is possible to perform a memory acquisition without saturating the processor. Despite the intense processor usage, our memory dumps remain coherent. We also showed that the acquisition is possible without overwhelming the disk or the network, and therefore it is possible to perform it over wireless embedded devices, mainly mobile phones. Our code footprint is considerably low, about 2200 lines of microvisor code.

References

[1] Stress-ng [pts/stress-ng]. (n.d.), 2019.

[2] Android, 2016. art and dalvik., 2020.

[3] Bpowershell empire. (n.d.)., 2020.

[4] En.wikipedia.org. (2019). rootkit.

[5] Amer Aljaedi, Dale Lindskog, Pavol Zavarsky, Ron Ruhl, and Fares Almari. Comparative analysis of volatile memory forensics: live response vs. memory imaging. In *2011 IEEE Third International Conference on Privacy, Security, Risk and Trust and 2011 IEEE Third International Conference on Social Computing*, pages 1253–1258. IEEE, 2011.

[6] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, volume 37, pages 164–177. ACM, 2003.

[7] Abderrahmane Benbachir and Michel R Dagenais. Hypertracing: Tracing through virtualization layers. *IEEE Transactions on Cloud Computing*, 2018.

[8] Bill Blunden. *The Rootkit arsenal: Escape and evasion in the dark corners of the system*. Jones & Bartlett Publishers, 2012.

[9] Andrew Case and Golden G Richard III. Memory forensics: The path forward. *Digital Investigation*, 20:23–33, 2017.

[10] Christoffer Dall and Jason Nieh. Kvm/arm: The design and implementation of the linux arm hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 333–348, New York, NY, USA, 2014. ACM.

[11] Rushita Dave, Nilay R Mistry, and MS Dahiya. Volatile memory based forensic artifacts & analysis. *Int J Res Appl Sci Eng Technol*, 2(1):120–124, 2014.

[12] Dan Farmer and Wietse Venema. *Forensic discovery*. Addison-Wesley Professional, 2009.

[13] Greg Hoglund and James Butler. *Rootkits: subverting the Windows kernel*. Addison-Wesley Professional, 2006.

[14] Eviatar Khen, Nezer J Zaidenberg, Amir Averbuch, and Evgeny Fraimovitch. Lgdb 2.0: Using lguest for kernel profiling, code coverage and simulation. In *2013 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, pages 78–85. IEEE, 2013.

[15] Michael Kiperberg, Roee Leon, Amit Resh, Asaf Algawi, and Nezer Zaidenberg. Hypervisor-assisted atomic memory acquisition in modern systems. In *International Conference on Information Systems Security and Privacy*. SCITEPRESS Science And Technology Publications, 2019.

[16] Ralph Langner. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security & Privacy*, 9(3):49–51, 2011.

[17] H Macht. Dalvikvm support for volatility, 2012.

[18] Aravind Menon, Jose Renato Santos, Yoshio Turner, G John Janakiraman, and Willy Zwaenepoel. Diagnosing performance overheads in the xen virtual machine environment. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 13–23. ACM, 2005.

[19] Deirde K Mulligan and Aaron K Perzanowski. The magnificence of the disaster: Reconstructing the sony bmg rootkit incident. *Berkeley Tech. LJ*, 22:1157, 2007.

[20] Digit Oktavianto and Iqbal Muhardianto. *Cuckoo malware analysis*. Packt Publishing Ltd, 2013.

[21] Bryan D Payne. Simplifying virtual machine introspection using libvmi. *Sandia report*, pages 43–44, 2012.

[22] Niels Penneman, Danielius Kudinskas, Alasdair Rawsthorne, Bjorn De Sutter, and Koen De Bosschere. Formal virtualization requirements for the arm architecture. *Journal of Systems Architecture*, 59(3):144–154, 2013.

[23] Niels Penneman, Danielius Kudinskas, Alasdair Rawsthorne, Bjorn De Sutter, and Koen De Bosschere. Formal virtualization requirements for the arm architecture. *Journal of Systems Architecture*, 59(3):144–154, 2013.

[24] Santiago M Pontiroli and F Roberto Martinez. The tao of .net and powershell malware analysis. In *Virus Bulletin Conference*, 2015.

[25] Zhengwei Qi, Chengcheng Xiang, Ruhui Ma, Jian Li, Haibing Guan, and David SL Wei. Forenvisor: A tool for acquiring and preserving reliable data in cloud live forensics. *IEEE Transactions on Cloud Computing*, 5(3):443–456, 2016.

[26] Eric D Rotvold, Donald R Lattimer, Michael J Green, Robert J Karschnia, and Marcos AV Peluso. Interface module for use with a modbus device network and a fieldbus device network, July 17 2007. US Patent 7,246,193.

[27] Rusty Russel. lguest: Implementing the little linux hypervisor. *OLS*, 7:173–178, 2007.

[28] Michel F Sanner et al. Python: a programming language for software integration and development. *J Mol Graph Model*, 17(1):57–61, 1999.

[29] Joe Sylve. Android mind reading: Memory acquisition and analysis with dmd and volatility. In *Shmoocon 2012*, 2012.

[30] Johannes Winter. Trusted computing building blocks for embedded linux-based arm trustzone platforms. In *Proceedings of the 3rd ACM workshop on Scalable trusted computing*, pages 21–30. ACM, 2008.

[31] Nezer J Zaidenberg and Eviatar Khen. Detecting kernel vulnerabilities during the development phase. In *2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing*, pages 224–230. IEEE, 2015.

[32] Nezer Jacob ZAIDENBERG. Hardware rooted security in industry 4.0 systems. *Cyber Defence in Industry 4.0 Systems and Related Logistics and IT Infrastructures*, 51:135, 2018.

[33] Lei Zhang, Lianhai Wang, Ruichao Zhang, Shuhui Zhang, and Yang Zhou. Live memory acquisition through firewire. In *International Conference on Forensics in Telecommunications, Information, and Multimedia*, pages 159–167. Springer, 2010.