Forensic Science International: Digital Investigation 49 (2024) 301758

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi

DFRWS USA 2024 - Selected Papers from the 24th Annual Digital Forensics Research Conference USA

In the time loop: Data remanence in main memory of virtual machines

Ella Savchenko [*], Jenny Ottmann, Felix Freiling

Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Department of Computer Science, Germany

ARTICLE INFO

Keywords:
Data remanence
Virtual machines
Memory forensics

ABSTRACT

Data remanence in the physical memory of computers, i.e., the fact that data remains temporarily in memory even after power is cut, is a well-known issue which can be exploited for recovering cryptographic keys and other data in forensic investigations. Since virtual machines in many aspects mimic their physical counterparts, we investigate whether data remanence is also observable in virtual machines. Using KVM as an example of virtualization technology, we experimentally show that it is common for a substantial amount of volatile data to remain in the memory of virtual machines after a reboot. In digital forensic analysis scenarios such as malware analysis using virtual machines, our observations imply high risks of evidence contamination if no precautions are taken. So while the symptoms of data remanence in virtual machines are similar to physical machines, the implications for digital forensic analysis appear very different.

1. Introduction

Random access memory (RAM) in modern computers is usually built out of large arrays of capacitors. If a capacitor is charged, then the corresponding bit in memory is set. Unfortunately, capacitors have leakage currents that eventually lead to a fully discharged state (representing an unset bit). So when power is cut on a normal PC, memory cells eventually lose their stored value. The time until a capacitor is fully discharged is called the *retention time*. Non-zero retention times are an expression of *data remanence* (Wyns and Anderson, 1989), i.e., the fact that data remains in memory even after power is cut.

While usually unwanted and often ignored on bare-metal systems, data remanence allows to recover the content of the main memory even after power was (shortly) cut from the device. This has been exploited for breaking the security of computing systems by Halderman et al. (2009) and Bauer et al. (2016): In their so-called *cold boot attacks*, the encryption key for relevant pieces of data is extracted from main memory shortly after the encryption or decryption was performed.

Today, an increasing fraction of computers does not run on bare-metal hardware but as a *guest* within a virtualized environment, called a virtual machine (VM). The technological concept behind this architecture is called *virtualization*, meaning the abstraction of physical hardware resources (Lombardi and Di Pietro, 2011). On the one hand, virtualization can solve the problem of the limitation of hardware resources by sharing them and running several VMs on the same physical machine, called a *host* (Smith and Nair, 2005). On the other hand, virtualization techniques support simple portability of programs, reusability of configurations, and dynamic loading of components (Arnold et al., 2005).

Virtualization today is technically supported by hardware extensions of modern processors (such as Intel VT-x). In short, they move the guest operating system (OS) to a less privileged execution level of the processor than usual and insert software, the *hypervisor*, at the highest privilege level that manages the guest(s). Access to the physical resources of a computer therefore undergoes an additional indirection. This holds also for the allocation of main memory: When a guest VM writes to memory, the physical pages it can access are regulated by the hypervisor. The de-allocation of memory of a VM, i.e., when it is switched off, is therefore an entirely different process from when a bare-metal system is switched off. But since virtual machines mimic their physical counterparts in many aspects, we ask the question: In what form does RAM data remanence occur in virtual machines when they are switched off? Answers to this question directly influence digital forensic processes regarding evidence handling when virtualized systems are involved since data remanence can both present new opportunities for information retrieval or pose risks of evidence contamination.

1.1. Related work

The lifetime of data in main memory of systems has long been a concern in security. For example, Chow et al. (2005) observed that on *running* systems, sensitive data like passwords may be recoverable in

* Corresponding author.
E-mail addresses: ella.savchenko@fau.de (E. Savchenko), jenny.ottmann@fau.de (J. Ottmann), felix.freiling@fau.de (F. Freiling).

https://doi.org/10.1016/j.fsidi.2024.301758

Available online 5 July 2024

memory for several days even after deallocation. The relevance of data that even survived a reboot, i.e., data remanence in electronics (Wyns and Anderson, 1989), was not discovered until the seminal work by Halderman et al. (2009). Subsequent work extended the technique of cold boot attacks to more involved and recent RAM technologies that use memory scrambling (Bauer et al., 2016; Yitbarek et al., 2017).

Until now, data remanence in VMs has been primarily investigated from the viewpoint of security. Under the heading of data remanence, several papers studied whether data existent in one instance of a VM could also be discovered in other instances of the VM. For example, Snyder and Jones (2019) found unique data in unallocated space of the file system on different VMs created from the same template in Amazon's AWS cloud. In a similar setting, Albelooshi et al. (2015) studied whether similar data leaks occur also in main memory. They found data such as usernames, passwords, and URLs in multiple VMs, but this also appears to have been caused by the use of the Amazon VM template from which all machines were created. In subsequent work, Albelooshi et al. (2015a) found traces of a guest VM's memory contents after it was terminated and deleted in the host's main memory.

In their work on the full-disk encryption system TRESOR, Müller et al. (2011) interestingly discovered that CPU registers of some virtual machines are not reset during a software reboot, which indicates a possible attack vector in tested virtual machines. However, they did not investigate whether the contents of RAM were still recoverable after reboot.

1.2. Contributions

In the context of this work, we define *data remanence in virtual machines* as the fact that volatile data in a VM is recoverable from *within* the VM after the VM was rebooted, either by a software reboot or by switching the VM off and on again. To the best of our knowledge, we are the first to study data remanence in virtual machines from a digital forensic perspective.

The main result of this paper is that we were able to show that data remanence in virtual machines exists but it is very different in nature from data remanence in bare-metal systems. But while the existence of data remanence to us came as a surprise, the differences to bare-metal systems are not surprising since the cause of data remanence in VMs must be the effect of software-only mechanisms, i.e., mechanisms that are unrelated to those that cause remanence on bare-metal systems.

Being a software-only effect, in principle, the mechanisms that cause data remanence in VMs can be investigated analytically, i.e., by studying the (source) code of the involved software systems. But because of the diversity and the complexity of modern virtualization technologies, in this paper, we take an *experimental* approach to study data remanence in VMs. More specifically, the contributions of this work are as follows:

1. We develop an automated testing method for the investigation of data remanence in the RAM of VMs. The basic idea is to write unique patterns into RAM and regularly test which ones are still retrievable in memory dumps. Our method is agnostic to the type of hypervisor and the type of OS of the guest.
2. For the case of the hypervisor KVM and Linux as guest OS, we show that data remanence to a large degree exists in cases where the guest performs a software reboot. If the guest is switched off and on, or if the guest is reverted to a clean snapshot (taken before patterns were written), we find no data remanence at all.
3. We characterize different types of patterns regarding data remanence and the circumstances under which they occur. For example, *lost* patterns are those that never occur in any subsequent memory dump and therefore resemble data loss similar to bare-metal systems. We also observe *phantom* patterns, i.e., patterns that may appear *only* after reboot and not before, a phenomenon not known from bare-metal systems that can be attributed to mechanisms in modern VMs that influence the time until a pattern actually reaches the memory of VMs.

The implications of our results to digital forensic analysis are subtle and affect all cases in which memory dumps are taken after the reboot of a VM. For example, if malware operates within a VM, it is highly probable that traces of its existence can be found in a memory dump taken after reboot. On the one hand, rebooting the VM is therefore a valid method to regain control over a VM without losing the ability to perform a detailed memory analysis of that malware sample.

On the other hand, if traces of malware (or other incriminating data) can be found in the memory of a VM, our findings imply that this may be due to data remanence from previous executions before the reboot of the VM. Data remanence therefore creates substantial risks for the contamination of digital evidence (Gruber et al., 2023) that analysts need to be aware of.

1.3. Outline

This paper is structured as follows: In the next section, background information about virtual machines and memory forensics is summarized. Then, the methodology and experimental setup are presented in Section 3. The results regarding data remanence are presented and discussed in Section 4. Finally, we discuss some limitations of our experiments in Section 5 and summarize our findings in Section 6.

2. Background

Waldspurger and Rosenblum (2012) define a VM as "software abstraction that behaves as a complete hardware computer, including virtualized CPUs, RAM, and I/O devices". An important component to manage VMs is the *Virtual Machine Monitor* (VMM) or *hypervisor* (Rosenblum and Garfinkel, 2005). The VMM is a software layer between the physical host machine and the VM, running on the host as shown in Fig. 1. The VMM controls the execution of VMs and provides an infrastructure for the usage of hardware resources. It also supports strong isolation of every VM, both from other VMs and the host, making it possible to execute multiple VMs with different OSs. Another feature of a VMM is the encapsulation of the VM's state. With this feature, a VM can be paused and resumed from the last state. This enables migration of VMs to a different host and supports cloning of VMs and sharing the complete system inside a virtual environment (Rosenblum and Garfinkel, 2005).

2.1. KVM

KVM is an open-source kernel module which turns the Linux kernel into a hypervisor that supports the execution of various VMs (Desai et al., 2013; KVM). In order to run and manage VMs, KVM works in cooperation with QEMU, an open-source software emulator. One of the

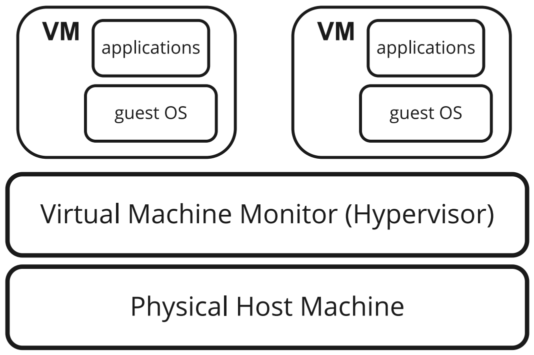

Fig. 1. Virtual Machine Monitor, modified from Al-Hawawreh (2017).

most commonly used features of QEMU is system emulation, i.e. implementing the emulation of CPU and other devices (QEMU Project Developers). The motivation to combine KVM and QEMU is to improve the performance of virtualization by executing guest instructions directly on the hardware as far as possible. For this part, KVM utilizes VT-x, the extension that is designed to support virtualization for x86 processors on the hardware level. QEMU, on its site, provides a translation for the instructions which cannot be executed directly (Goto, 2011). For the communication between KVM and QEMU, the kernel creates a device node called *dev/kvm*, as shown in Fig. 2. This device is used by QEMU, which runs as a regular user process, to create a VM and instruct KVM about required hypervisor functions, such as I/O requests. The QEMU process will be created for each VM separately and can be managed through *virt-manager* (Kivity et al., 2007; Goto, 2011). An example usage of virt-manager/virsh is changing the state of a VM, such as starting, powering off, or restarting. The implementation of virt-manager/virsh relies on *libvirt*, a toolkit and library for managing virtualization platforms (libvirt).

2.2. Digital memory forensics and RAM data

Typically, running processes use RAM to store data at some point in time. Therefore, this type of memory contains a lot of important information about activities on a system, especially from a forensic perspective. Besides information about active processes, hidden or stopped processes can be found during the memory analysis. This can, for example, be used for identifying malicious software. Also, browser data and network connections can be recovered. This data can include the content of conversations or downloaded files. Other relevant information which can be found are passwords or cryptographic keys. Those are often stored unencrypted in the RAM and can be used to decrypt other files or data on a system. Further examples of useful data in the RAM are information about open files, logged-in users, or the content of the clipboard (Hausknecht et al., 2015).

There are two main approaches for obtaining RAM data of physical computers in the form of a *memory dump*. One approach is to create a copy of the data with software tools, many of which are free of cost. However, the execution of those tools on a computer uses the same memory as every other process in the system. Therefore, this approach leaves some traces in memory too, which needs to be considered during the investigation. Another method is making a copy of the RAM contents with a special peripheral. This avoids interaction with other computer components and ensures integrity without any unintentional changes but the additional hardware leads to increased costs (Hausknecht et al., 2015). On virtualized systems, the integrity of the memory contents can also be ensured since VMMs have full access to the memory assigned to the guest. They can pause the execution of the guest and then copy the memory assigned to it without making any internal changes.

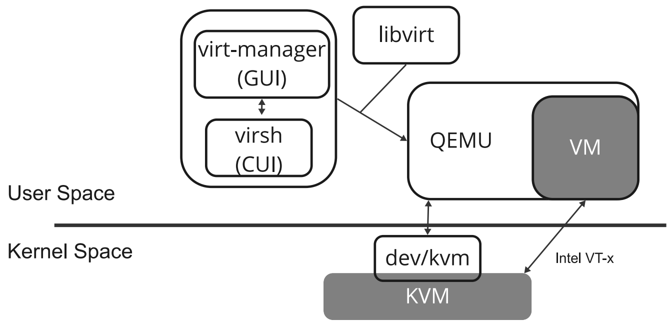

Fig. 2. KVM and QEMU, modified from Goto (2011).

3. Methodology and experimental setup

We conduct our experiments on a physical computer (Fujitsu ESPRIMO P900) with an Intel Core i7-2600 CPU (x86, 4 cores, 8 threads), 16 GB of RAM, 500 GB of hard-disk capacity, and Ubuntu 22.04.1 LTS (64-bit) as the OS. The firmware is BIOS. The host supports VT-x hardware-assisted virtualization.

The kernel version for the host is 6.5.0–21 generic, and for the guest 5.4.0–84 generic. For the KVM hypervisor, we run the QEMU emulator (version 6.2.0) and use Virtual Machine Manager (version 4.0). We have installed virsh (version 8.0) for the management of the KVM VMs. The VM has Ubuntu 18.04, 64-bit as the OS, 1 CPU and 2048 MB RAM.

To minimize possible side-effects of caching regarding the evaluation of data remanence we chose *writethrough* caching mode in KVM for our experiments. In a pre-study, we investigated the occurrence of remanence across multiple VMs running on the same physical host. Here we can not claim the spreading of data and data remanence from one VM into another. The presented setup describes therefore the investigations of data remanence in the context of multiple instances of the same VM.

The experimental framework as well as the experimental data are available online.[1]

3.1. Data investigated for remanence

We investigate data remanence in a VM by evaluating the occurrence of patterns in the dump of its RAM. These patterns, abbreviated as *wtime*, are intentionally written into memory. The structure of a wtime pattern consists of four consecutive elements, each of size 64 Bits. The first and the last of these elements form a frame with a constant value which we denote as *w-prefix*. In our experiments, the w-prefix had the value *0xDEADFA11C0FEBABE*. The second element represents the pattern's number in the writing order, which we call η. The third element is a Unix timestamp in milliseconds in hexadecimal representation. This timestamp corresponds to the time at which the pattern was created. Since only one wtime pattern is created every 10 ms, each wtime pattern is unique by its timestamp. In our experiments, we measure how many such wtime patterns can be found in successive memory dumps of the VM's RAM.

3.2. Methodology for investigating data remanence

Investigating data remanence implies the presence of data inside of a VM and the ability to discover this data again within the VM after a restart. So overall, our experimental method follows three main steps:

1. Write wtime patterns into the RAM of the VM. This step is called *writeDATA*.
2. Restart the VM (either via reboot or poweroff/poweron).
3. Search for wtime patterns in memory dumps of the VM. This step is called *searchDATA*.

We now explain these three steps in more detail.

In the *writeDATA* step, we use a small C program, called writeDATA, to write wtime patterns to the VM's RAM. The program runs for η iterations. In each iteration, one wtime pattern is created as described in Section 3.1. To avoid false positives by storing the *w-prefix* pattern in memory within the code or data of the C program, the w-prefix is created by XOR of two hexadecimal values at run time.

For each wtime pattern, we allocate one separate memory page of size 4096 kB. We do not influence the physical addresses to which the pages are mapped. Therefore, the patterns are spread across the RAM according to the OSs memory allocation strategy. To ensure the uniqueness of timestamps and to avoid race conditions, at the end of

[1] https://github.com/ella-savchenko/InTheTimeLoop.git.

each iteration, the program waits for ϵ time before the next wtime pattern is written. After writing a total of η patterns, the program waits for δ time to ensure that all regular system and caching activity has normalized, and then terminates. Fig. 3 summarizes the sequence of steps executed at this stage of the experiment in pseudocode.

Restarting the VM can be done in two different ways, either by using a software reboot from the VMM or by explicitly powering off and then powering on the VM. Additionally, the VM can be reverted to a snapshot that was taken before the patterns were written into the memory. Two classes of memory dumps are acquired using the VMM at well-defined points in time: memory dumps W_i (W for "written patterns") are acquired before the restart/revert and memory dumps R_i (R for "reboot") after the restart/revert. We describe these dumps in more detail below.

In the *searchDATA* step, the sequence of memory dumps is then systematically analyzed for occurrences of wtime patterns. The memory dumps are captured using the command-line interface of Virtual Machine Manager. The search procedure uses the w-prefix at the beginning of a memory page as search pattern. If the pattern is found twice at the beginning of a page with two 64 bit values between, we consider this a *hit*. To avoid false positives, we implemented components for automated logging of the timestamps within the host such that the timestamps in the dump could be cross-checked using that log.

Considering the VM RAM, we aim to overwrite at least 40 % of free memory with wtime patterns, but not much more to avoid swapping. In our setup with 2048 MB, ≈800 MB remains free for activities other than regular OS usage. In our experimental setup, we chose the number of iterations $\eta = 100k$, overwriting 100k memory pages, which corresponds to ≈45 % of free memory. We set the wait time at the end of each iteration, ϵ, to 10 ms, and the wait time after the completion of all iterations, δ, to 10 min.

We implemented the steps presented above as an automated investigation process which can be executed for a variable number of experiments H. The structure of the overall sequence of experiments is given as pseudocode in Fig. 5. We now give some more details on how individual experiments were performed. An experiment follows the schedule given graphically in Fig. 4.

At the beginning of each experiment, we start the VM using the VMM interface and execute writeDATA inside the VM to write wtime patterns into the RAM of the VM. Halfway through writeDATA, we create the first memory dump W_1 in which we expect to observe about half of the wtime patterns. The process of writing a total of $\eta = 100.000$ patterns into memory (with $\epsilon = 10$ ms waiting time each) lasts approximately 1000 s so that at "halftime", i.e. when W_1 is taken, 500 s have passed since the start of writeDATA. Memory dump W_1 is basically a plausibility check to verify whether writeDATA is executed as expected.

After all η wtime patterns are written into the RAM, writeDATA waits for another $\delta = 10$mins before terminating. During this period we perform further memory dumps, W_2 and W_3. Memory dump W_2 is taken 110 s after finishing writing patterns, and W_3 5 min after W_2. About 1 min after the termination of writeDATA, we perform the memory dump W_4. Two further dumps, W_5 and W_6, are taken in 5-min intervals. After performing all W_i dumps, we wait for another minute and then reboot

```
for $j = 1$ to $\eta$ do
    allocate new memory page $p$
    write w-prefix to $p$
    write $j$ to $p$
    write timestamp in ms to $p$
    write w-prefix to $p$
    wait $\epsilon$ time
end for
wait $\delta$ time
```

Fig. 3. Pseudocode of *writeDATA* phase of experiment.

the VM. One minute after the VM has been started, we perform the dump R_1. Then, each in 5-min intervals, further dumps R_2 and R_3 follow. After performing the last dump, R_3, we wait for another minute, and power off the VM.

The structure of the wtime patterns is used to search for patterns written across different writeDATA iterations η and experiments H in the searchDATA stage. The search is implemented via grep regular expression on the VM RAM dump files. With a Perl regular expression, we search for the w-prefix frame and additional parts of patterns inside this frame, such as iteration number η and timestamp. This specific structure enables us to find and distinguish patterns from different runs. As discussed in Section 3.1, we assign a unique identifier for each pattern to track the origin of the data remanence to the experiment iteration where the data is deliberately written into the RAM. Since at most one pattern is written per millisecond, we can consider the time of writing to be unique for each pattern. It seems unlikely for such a specific structured sequence of bits to appear frequently in memory by chance, which we also verify experimentally. During our experiments with $H = 125$ we did not observe a bit sequence which appeared by chance in the form of the wtime pattern.

4. Evaluating data remanence

We now explain the results of our experiments in detail, starting with the data generation on which our analysis of data remanence builds.

4.1. Data generation

During the experiments, extensive logging was performed to generate data and to be able to understand the measurements and avoid errors.

The collected information includes an experiment number, its start and end times, and the results of actions to control a VM representing the state of the VM in a period of time. We also recorded information about the original data written into memory in the form of timestamps of wtime patterns. This block serves the purpose of controlling whether writeDATA was executed successfully and to support the further classification of data remanence.

4.2. No remanence after power-off

One of the parameters we evaluated was whether the VM was rebooted or powered off and on again (see Fig. 5). This parameter has a very noticeable effect on the results since we observed no remanence at all in the latter case: After powering off the VM and restarting it anew, not a single wtime pattern could be found in memory dump R_1.

This behaviour is most likely caused by the different ways the VMM handles both cases.

Across our experiments, we observed that the PID for the underlying VM process remains the same after rebooting the VM and changes when the VM is powered off and on again. This indicates that the VM appears to keep the same address space after reboot but is allocated a new address space when it is powered off and on.

Data remanence therefore appears to be a side effect of this behaviour of the hypervisor. Thus, even if writeDATA's memory is freed after its termination and the VM is rebooted, the allocated guest memory still appears to be in use from the viewpoint of the hypervisor. Freeing memory within the guest does not clear the memory. In contrast, allocation of memory in a new address space will usually result in cleared memory.

4.3. No remanence after reverting to a snapshot

Another parameter we investigated was whether the VM was restarted or reverted to a "clean" snapshot (see Fig. 5). This snapshot was taken at the beginning of each iteration, after powering off the VM,

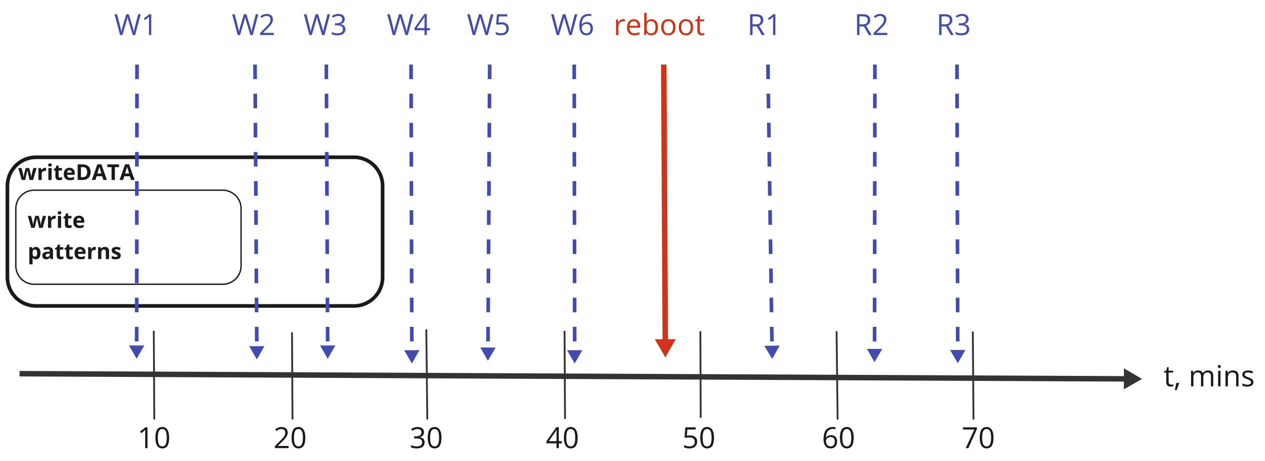

Fig. 4. Memory dump acquisition over time. W_1 is performed at half of the time while writeDATA is writing patterns into the RAM. W_2 is created right after writeDATA has finished writing patterns into the RAM but has not yet terminated. 5 min later, W_3 is also taken while writeDATA is still running. W_4, W_5, and W_6 are performed after writeDATA has been terminated. After the VM has been rebooted, R_1, R_2, and R_3 are performed, writeDATA is *not* executed.

```
for $k = 1$ to $H$ do
    poweron VM
    cobegin                              ▷ write patterns in memory
        execute writeDATA in VM          ▷ according to Fig. 3
    ||
        generate memory dumps $W_1$–$W_3$
    coend
    generate memory dumps $W_4$–$W_6$
    either reboot VM or poweroff/poweron VM or
         revert to a snapshot
    generate memory dumps $R_1$, $R_2$, $R_3$
    poweroff VM
end for
```

Fig. 5. Pseudocode of the overall experimental design. The keywords **cobegin**, || and **coend** capture concurrent/parallel activity of data generation and data extraction (see also Fig. 4).

then powering on the VM, and before writing patterns into the memory.

Rather unsurprisingly, we did not observe any remanence in this case. This confirms good security practices in malware analysis where reverting to a clean snapshot is a standard procedure in dynamic analysis.

4.4. Remanence after reboot

In contrast to the case of power-off/on and reverting to a snapshot, we could observe a substantial amount of remanence after rebooting the VM in our experiments. We summarize the experimental results in Table 1. Since we generated 6 memory dumps before reboot (W_1–W_6) and 3 memory dumps after reboot (R_1–R_3), we use the notation D_i to consistently refer to them, where i is a number between 1 and 9 denoting the line in Table 1 referring to that memory dump. For instance, D_4 refers to W_4 and D_9 to R_3.

Table 1
Distribution of the number of patterns across different memory dumps over time for $H = 125$ experiment runs.

i	D_i	mean	SD	median	MAD	min	max	IQR
1	W_1	43503	1028	43666	304	37689	44559	155
2	W_2	97887	1461	98321	449	91929	98388	26
3	W_3	97918	1448	98322	420	92426	98388	26
4	W_4	94875	1594	98322	462	89950	98388	27
5	W_5	97371	3765	98319	966	63600	98388	28
6	W_6	96585	5217	98315	1748	63594	98388	50
7	R_1	94389	7051	97994	3805	60688	98329	6055
8	R_2	92074	12573	97117	5416	7828	98286	6645
9	R_3	85096	16997	90013	8219	0	96686	8219

Overall, $H = 125$ experiments were performed for each memory dump D_i. Let $P_{i,j}$ denote the number of patterns that were recovered for memory dump D_i in experiment j. The table shows some statistical properties of the measurements $P_{i,1}$ to $P_{i,125}$ for each memory dump D_i (the formulas used for the calculations are given in Appendix A as a reference). The *mean* and *median* give an impression of the average or middle value of the measurements together with measures for the variance, namely *standard deviation* (SD) and *median absolute deviation* (MAD). The table also shows the minimum and maximum value of patterns found in D_i as well as the *interquartile range* (IQR) describing the tendency of the 3rd and the 1st quartile of pattern findings during the experiments. All of these measurements are also depicted in Fig. 6. The areas marked as SD/MAD visualize the average spread along the mean and median lines respectively. The IQRs are represented as boxplots with whiskers extending up to a maximum of 1,5 times the size of the IQR. The whiskers show the common distribution of patterns beyond the IQR range, which are not identified as outliers. These are illustrated as red markers.

Looking at the data in Table 1, recall that during the writeDATA step, we attempt to write 100.000 wtime patterns into the RAM of the VM. We denote by F (for 'forecast') the hypothetical set of *all* wtime patterns written in an experiment. The set F is therefore a reference for the limit of patterns that may be found in a dump. It can also serve as a plausibility check for spurious wtime patterns (those not contained in F).

Fig. 6 shows the development of the number of found patterns over time. The low numbers of found patterns in W_1 results from this memory dump being taken "half way through" the writeDATA step of the experiment. The number of found patterns has a rather low variance with a few outliers towards lower values. Memory dump W_2 is performed 110 s after writeDATA has finished writing patterns into the RAM of the VM. The maximum number of patterns is found in W_3 which is very close to the maximum possible value of $|F| = 100.000$.

Overall, we can observe a rather stable number of retrieved patterns before reboot, i.e., even in memory dump W_6 still more than 96 % of all patterns can still be found on average with the median being constantly larger than the average. This is consistent with what Chow et al. (2005) observed on running bare-metal systems.

After reboot the number of patterns found decreases substantially over time. Even more so, the variability in the measurements clearly increases with severe outliers towards low values. This can be seen by

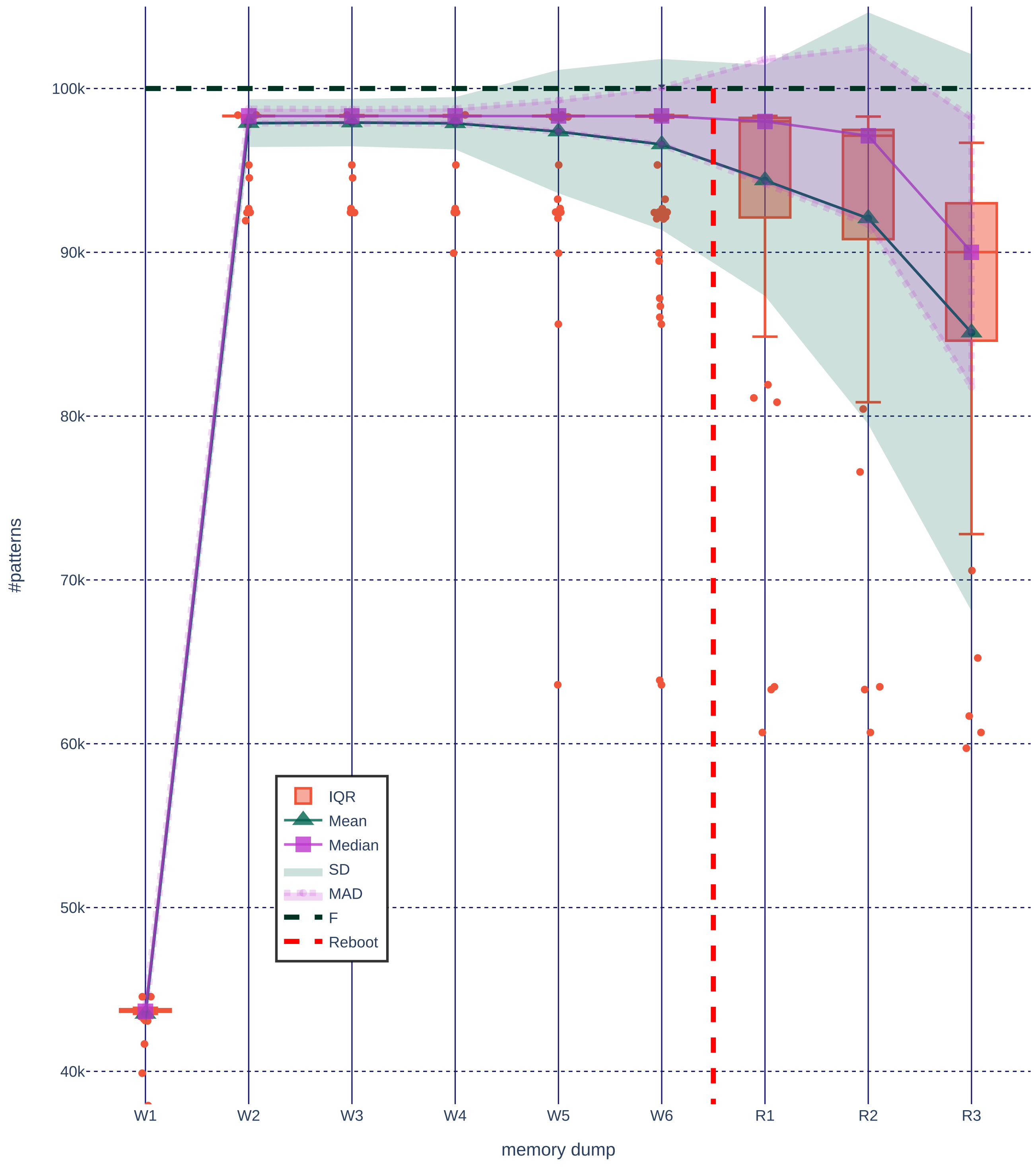

Fig. 6. Development of numbers of recovered patterns across the sequence of memory dumps over time. Note the visualisation is presented with the lower threshold of 38.000 patterns for a better trend overview. See Table 1 for minimum and maximum values.

the values of the median that remains above 90.000 even in R_3 while the minimum values are less than 8.000 (for R_2) and even 0 (for R_3).

We notice the decrease of the number of patterns also in a long-term observation. To assess this, the observed time after reboot and before powering off the VM was extended to 72 h. After investigation of the memory dump R_3, we examine additional memory dumps, taken every 15 min without changing the state of the VM. We denote these memory dumps $R-(h)$, where h stands for each full hour since R_3.

We conducted $H=3$ experiment runs with the mentioned parameters, the results of which are shown in Fig. 7. In the first 12 h, we notice a rapid decrease in the number of found patterns, from 90.000 to modestly higher than 20.000 at its median. After that, the number of patterns persists at a stable level for a prolonged period, but still slowly decreases. Even after 72 h the median of found patterns remains slightly under 20.000.

4.5. *Origins of patterns*

Apart from the evaluation of the overall number of patterns, we also investigate the origins of the found patterns. Since each wtime pattern is

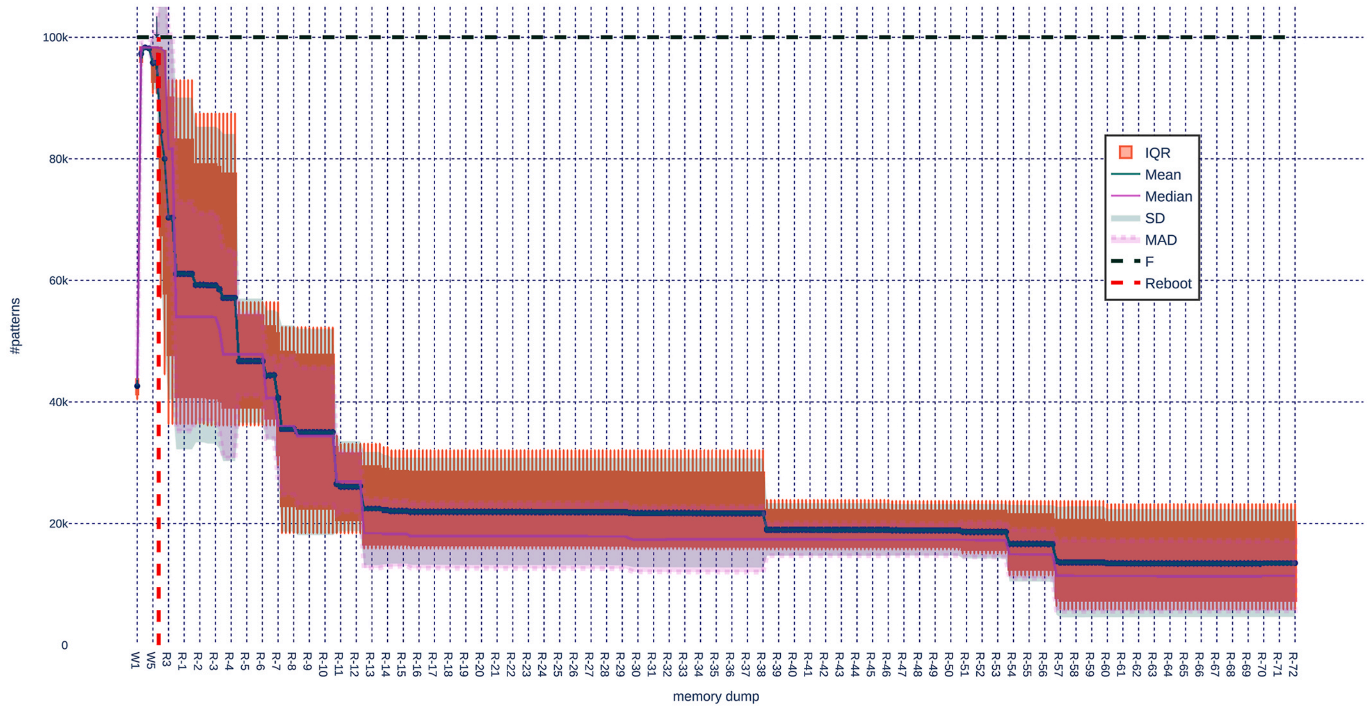

Fig. 7. Development of numbers of recovered patterns across the sequence of memory dumps over 72 h. *R-h* represents memory dump after each full past hour *h* since R_3.

unique by its timestamp, we can follow its appearance across different memory dumps, and therefore over time. Overall, we did not find any patterns in one experiment that actually originated in another experiment. This corresponds to the findings described above that there was no remanence across power-off/on of the VM. Furthermore, we never found any *spurious* patterns, i.e., all patterns we found actually were valid wtime patterns originating from that experiment.

Interestingly we *never* obtained a *perfect* memory dump, i.e., a dump in which the ideal set F of wtime patterns was observable. This implies that it is hard to find an optimal spot at which to maximize the number of found patterns, since memory dumps either occur too early (written patterns have not yet reached memory) or too late (patterns are already being deallocated). In our measurements, memory dump W_3 which was performed approximately 7 min after writeDATA has finished, has the peak value of patterns found in all dumps. After reaching this peak, the set of patterns found in subsequent memory dumps was always a strict subset of the previous memory dump, i.e., starting with $i = 3$ we found that $D_i \supset D_{i+1}$, even after reboot. This means that patterns monotonously vanish from memory without reappearing.

4.6. Proper remanence

Following our definition, (proper) remanence is manifested in patterns that can be discovered after rebooting the VM. The maximum amount of remanence can therefore be observed in the first memory dump R_1 after reboot.

Within this memory dump, on average 94.389 patterns out of 100.000 could be retrieved, with a median of 97.994. This corresponds to approximately 99.66 % of the median of the “best” memory dump W_3 before reboot. While the IQR is very low before reboot, it increases substantially after reboot identifying stark fluctuations in the remanence observations. This means that though the median for remanence is very large, there may be experiments in which only marginal remanence occurs. These observations show that remanence occurs frequently. However, fluctuations are common and the number of observed patterns decreases drastically over time.

4.7. Lost and phantom patterns

As noted above, we never acquired a memory dump with an optimal set of patterns that were previously written. So there is also a number of patterns that were written but actually never occur in any memory dump D_i. We call such patterns *lost*. More specifically, we define the number of *lost patterns until memory dump D_i* as follows:

$$\text{lost}(i) = |F \setminus \cup \{D_1, \dots, D_i\}|$$

Since the number of observed patterns increases until W_3, the number of lost patterns decreases until then and remains constant afterwards. The results of calculating the median, minimum and maximum of lost(i) are shown in Table 2. The fluctuation of lost patterns is moderate and remains rather stable across all memory dumps. The median at the time of creating a memory dump W_1 lies at 56353 patterns, for W_2 it is at 1681 patterns, and it is 1678 thereafter. This indicates that some lost patterns from W_1 and W_2 could be found in later dumps. On the other hand, we were not able to discover any patterns which were lost at the time of performing dump W_3 in subsequent memory dumps. These patterns are lost eternally.

As mentioned above, remanence appears to decrease with time and no patterns appear in remanence that have not been observed before reboot. However, comparing the patterns contained in R_1 with the patterns in the memory dumps before reboot, there are cases where we can find patterns in R_1 which did not occur in some memory dump W_i before reboot. We call these patterns *phantom patterns*. Formally, we define the number of *phantom patterns regarding memory dump W_i* as

$$\text{phantom}(i) = |R_1 \setminus W_i|.$$

Table 2
Amounts of lost lost(i) patterns over $H = 125$ experiments.

	median	MAD	min	max	IQR
W_1	56353	289	56272	62309	46
W_2	1681	469	1611	8070	26
$D_i, i \geq 3$	1678	435	1611	7574	25

Table 3
Number of phantom patterns phantom(i) over $H = 125$ experiment runs.

	median	MAD	min	max	IQR
W_1	54650	1316	41244	55349	148
W_2	0	25	0	561	0
$W_i, i \geq 3$	0	0	0	0	0

The measurements of phantom patterns are shown in Table 3. While we did not find any phantom patterns regarding $W_3, \ldots, W_6$, we do observe some phantom patterns when comparing with the patterns found in W_1 and W_2. Regarding W_1, on average about 54650 phantom patterns were found (with moderate fluctuation at $IQR = 148$). The phantom patterns concerning W_2 appear only rarely (in 9 out of 125 performed experiments). Concerning the distribution of phantom patterns across these 9 experiments, we identify their median at 270 patterns, with corresponding MAD at 153 patterns and IQR at 308 patterns. The minimum value across the experiments with phantom patterns is 43 patterns, and the maximum at 561.

5. Limitations

In this work, we deliberately chose an experimental approach for studying the effect of only a limited set of variables on data remanence in VMs. Certainly, our methodology can be applied to additional parameters, hypervisors and OSs. For example, the number of CPUs and the RAM size could be varied, or optimizations like memory ballooning enabled. Since the approach does not rely on insight into the implementation of the hypervisor or OS, it can also be applied to closed-source implementations.

The evaluation results presented in this paper are limited to the KVM hypervisor and the chosen experimental setup, such as writethrough caching mode. Other hypervisors or configurations of VMs may cause different effects. We also observe stark fluctuations in the occurrence of data remanence. Performing more experiment runs can influence the average for remanence we refer to. Future evaluations could explore whether the variations in the observed amount of remanence are connected to settings of the hypervisor or other variables of the experiment.

6. Conclusion

During our investigations with KVM VMs, we observed data remanence for different instances of the same VM after performing a software reboot. Depending on the dump time, data remanence occurs with a large fluctuation and on average for ≈99 % of written data. We identify the origin for this remanence in the previous instance of the VM within one experiment run. This is likely connected to the way in which the hypervisor handles the reboot. Since the process running the VM is not terminated during the reboot, no new address space is allocated for the VM. Therefore, since the hypervisor apparently does not overwrite the memory, data remanence can occur.

From a forensic perspective, this kind of remanence in VMs is of peculiar interest in testing scenarios. When performing analyses that rely on identifying artefacts of programs that are part of unallocated memory, i.e., when carving for data structures, after a reboot artefacts from a previous instance could be found. Therefore, while virtual environments offer benefits for such analyses, due to data remanence, rebooting the VM after analyzing one sample might introduce contamination to the results of a subsequent analysis performed on the rebooted VM. As we saw with the phantom patterns, depending on the runtime of the VM after performing an initial analysis, data structures might even be retrievable after the reboot that were not found before. The extent of retrievable memory contents of terminated programs likely also depends on the amount of active processes before the reboot. For the hypervisor we used in the experiments, remanence can be avoided by turning the VM off and starting a new instance instead of rebooting the VM.

Our evaluation also confirms that determining the best time to take a memory dump might be tricky. Thus, if enough resources (especially time and storage) are available it might be recommendable to take multiple memory dumps. For example, in our setup, a memory dump taken 5 min after the termination of writeDATA contained more patterns than one taken directly after its termination. Naturally, the exact timing depends on the encountered system setup and running programs, but taking multiple memory dumps before potentially turning off the VM can result in more data regarding still running or recently terminated programs. As our results show, this data might also still be available after rebooting the VM.

For anti-forensics scenarios, we recommend using encryption methods such as AMD SEV-SNP (AMD, 2020) or Intel TDX (Cheng et al., 2023) to ensure the confidentiality of the VM's data, even if some data remanence occurs. Also, an additional patch to force hypervisors to reallocate memory space can reduce risks of data remanence.

CRediT authorship contribution statement

Ella Savchenko: Conceptualization, Investigation, Methodology, Software, Writing - Original Draft, Writing - Review & Editing. **Jenny Ottmann:** Conceptualization, Supervision, Writing - Review & Editing. **Felix Freiling:** Conceptualization, Supervision, Writing - Review & Editing.

Acknowledgments

We thank Christian Eichenmüller, Julian Geus, Immanuel Lautner, Ralph Palutke, Gaston Pugliese, Volkmar Sieh, and Michael Tielemann for helpful discussions. Work was supported by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) as part of the Research and Training Group 2475 "Cybercrime and Forensic Computing" (grant number 393541319/GRK2475/1-2019).

Appendix A. Definitions of statistical metrics

As a reference, we provide here the standard definitions of the statistical metrics we used for data analysis in Section 4.

The set F is the hypothetical set of all patterns recoverable from a memory dump D_i. Overall, $H = 125$ experiments were performed for each memory dump D_i. Let $P_{i,j}$ denote the number of patterns that were recovered for memory dump D_i in experiment j.

The *mean* of the number of patterns for memory dump D_i is calculated as

$$\overline{P_i} = \frac{1}{H}\sum_{j=1}^{H} P_{i,j}$$

and the *standard deviation* (SD or σ) of patterns for memory dump D_i is

$$\sqrt{\frac{\sum_{j=1}^{H}\left(P_{i,j}-\overline{P_i}\right)^2}{H-1}}$$

Let $S_1, \ldots, S_H$ be the sorted sequence of values $P_{i,1}, \ldots, P_{i,H}$ for memory dump D_i. The *median* (or *middle value*) of the number of patterns for emory dump D_i then is

$$\mathtt{median}(P_i) = S_{\frac{H+1}{2}}.$$

The *median absolute deviation* (MAD) then is calculated as the median of the deviations of $P_{i,j}$ from the median of P_i, i.e.,

$$\text{median}(|P_i - \text{median}(P_i)|).$$

As further measures of variability, for each D_i we compute the *interquartile range* (IQR) describing the tendency of the 3rd and the 1st quartile of pattern findings during experiments. These are defined as follows:

$$Q_1 = \text{median}\left(\frac{1}{2}\left(S_1, S_2, \ldots, S_{\frac{H}{2}}\right)\right)$$

$$Q_3 = \text{median}\left(\frac{1}{2}\left(S_{\frac{H}{2}+1}, S_{\frac{H}{2}+2}, \ldots, S_H\right)\right)$$

$$\text{IQR} = Q_3 - Q_1$$

References

Al-Hawawreh, M.S., 2017. SYN flood attack detection in cloud environment based on TCP/IP header statistical features. In: 2017 8th International Conference on Information Technology (ICIT). IEEE, pp. 236–243.

Albelooshi, B., Salah, K., Martin, T., Damiani, E., 2015a. Experimental proof: data remanence in cloud VMs. In: 2015 IEEE 8th International Conference on Cloud Computing. IEEE, pp. 1017–1020.

Albelooshi, B., Salah, K., Martin, T., Damiani, E., 2015. Inspection and deconfliction of published virtual machine templates' remnant data for improved assurance in public clouds. In: 2015 IEEE/ACS 12th International Conference of Computer Systems and Applications (AICCSA). IEEE, pp. 1–4.

AMD, 2020. Strengthening VM Isolation with Integrity Protection and More, vol. 53. White Paper, pp. 1450–1465. January.

Arnold, M., Fink, S.J., Grove, D., Hind, M., Sweeney, P.F., 2005. A survey of adaptive optimization in virtual machines. Proc. IEEE 93, 449–466.

Bauer, J., Gruhn, M., Freiling, F.C., 2016. Lest we forget: cold-boot attacks on scrambled DDR3 memory. Digit. Invest. 16 (Suppl. ment), S65–S74. https://doi.org/10.1016/j.diin.2016.01.009.

Cheng, P.C., Ozga, W., Valdez, E., Ahmed, S., Gu, Z., Jamjoom, H., Franke, H., Bottomley, J., 2023. Intel TDX Demystified: A Top-Down Approach. arXiv Preprint arXiv:2303.15540.

Chow, J., Pfaff, B., Garfinkel, T., Rosenblum, M., 2005. Shredding your garbage: Reducing data lifetime through secure deallocation. In: McDaniel, P.D. (Ed.), Proceedings of the 14th USENIX Security Symposium. Baltimore, MD, USA, July 31 - August 5, 2005, USENIX Association. URL: https://www.usenix.org/conference/14th-usenix-security-symposium/shredding-your-garbage-reducing-data-lifetime-through.

Desai, A., Oza, R., Sharma, P., Patel, B., 2013. Hypervisor: a survey on concepts and taxonomy. Int. J. Innovative Technol. Explor. Eng. 2, 222–225.

Goto, Y., 2011. Kernel-based virtual machine technology. Fujitsu Sci. Tech. J. 47, 362–368.

Gruber, J., Hargreaves, C.J., Freiling, F.C., 2023. Contamination of digital evidence: understanding an underexposed risk. Forensic Sci. Int. Digit. Investig. 44, 301501 https://doi.org/10.1016/J.FSIDI.2023.301501. URL.

Halderman, J.A., Schoen, S.D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J.A., Feldman, A.J., Appelbaum, J., Felten, E.W., 2009. Lest we remember: cold-boot attacks on encryption keys. Commun. ACM 52, 91–98.

Hausknecht, K., Foit, D., Burić, J., 2015. RAM data significance in digital forensics. In: 2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO). IEEE, pp. 1372–1375.

Kivity, A., Kamay, Y., Laor, D., Lublin, U., Liguori, A., 2007. KVM: the Linux virtual machine monitor. In: Proceedings of the Linux Symposium, Dttawa, Dntorio, Canada, pp. 225–230.

KVM, . KVM main page. URL: https://www.linux-kvm.org/page/Main_Page.online; accessed 2023-January-18.

libvirt, . libvirt virtualization api. URL: https://libvirt.org/.online; accessed 2023-January-18.

Lombardi, F., Di Pietro, R., 2011. Secure virtualization for cloud computing. J. Netw. Comput. Appl. 34, 1113–1122.

Müller, T., Freiling, F.C., Dewald, A., 2011. TRESOR runs encryption securely outside RAM. In: 20th USENIX Security Symposium (USENIX Security 11).

QEMU Project Developers, About QEMU: QEMU's documentation.. URL: https://www.qemu.org/docs/master/. online; accessed: 2023-October-7.

Rosenblum, M., Garfinkel, T., 2005. Virtual machine monitors: current technology and future trends. Computer 38, 39–47. https://doi.org/10.1109/MC.2005.176.

Smith, J.E., Nair, R., 2005. The architecture of virtual machines. Computer 38, 32–38.

Snyder, B.L., Jones, J.H., 2019. Determining the effectiveness of data remanence prevention in the AWS cloud. In: 2019 7th International Symposium on Digital Forensics and Security (ISDFS). IEEE, pp. 1–6.

Waldspurger, C., Rosenblum, M., 2012. I/O virtualization. Commun. ACM 55, 66–73.

Wyns, P., Anderson, R., 1989. Low-temperature operation of silicon dynamic random-access memories. IEEE Trans. Electron. Dev. 36, 1423–1428. https://doi.org/10.1109/16.30954.

Yitbarek, S.F., Aga, M.T., Das, R., Austin, T.M., 2017. Cold boot attacks are still hot: security analysis of memory scramblers in modern processors. In: 2017 IEEE International Symposium on High Performance Computer Architecture, HPCA 2017, Austin, TX, USA, February 4-8, 2017. IEEE Computer Society, pp. 313–324. https://doi.org/10.1109/HPCA.2017.10.