Introducing the Temporal Dimension to Memory Forensics

FABIO PAGANI, Eurecom, France
OLEKSII FEDOROV, Igor Sikorsky Kyiv Polytechnic Institute, Ukraine
DAVIDE BALZAROTTI, Eurecom, France

Kickstarted by the Digital Forensic Research Workshop (DFRWS) conference in 2005, modern memory analysis is now one of most active areas of computer forensics and it mostly focuses on techniques to locate key operating system data structures and extract high-level information. These techniques work on the assumption that the information inside a memory dump is consistent and the copy of the physical memory was obtained in an atomic operation.

Unfortunately, this is seldom the case in real investigations, where software acquisition tools record information while the rest of the system is running. Thus, since the content of the memory is changing very rapidly, the resulting memory dump may contain inconsistent data. While this problem is known, its consequences are unclear and often overlooked. Unfortunately, errors can be very subtle and can affect the results of an analysis in ways that are difficult to detect.

In this article, we argue that memory forensics should also consider the time in which each piece of data was acquired. This new *temporal dimension* provides a preliminary way to assess the reliability of a given result and opens the door to new research directions that can minimize the effect of the acquisition time or detect inconsistencies. To support our hypothesis, we conducted several experiments to show that inconsistencies are very frequent and can negatively impact an analysis. We then discuss modifications we made to popular memory forensic tools to make the temporal dimension explicit during the analysis and to minimize its effect by resorting to a *locality-based* acquisition.

CCS Concepts: • **Applied computing** → **System forensics**;

Additional Key Words and Phrases: Memory forensics, temporal dimension, memory acquisition, atomicity

ACM Reference format:
Fabio Pagani, Oleksii Fedorov, and Davide Balzarotti. 2019. Introducing the Temporal Dimension to Memory Forensics. *ACM Trans. Priv. Secur.* 22, 2, Article 9 (March 2019), 21 pages.
https://doi.org/10.1145/3310355

1 INTRODUCTION

Memory analysis is rapidly becoming one of the most important components of digital forensics. Over the past 10 years, it evolved from the use of simple carving techniques designed to grab string-like objects from raw memory dumps to a complex research field whose goal is to correctly recover the semantic of a large amount of information stored in memory. For this reason, to date, most of the research in this area has focused on techniques to overcome the *semantic gap* and reconstruct a faithful picture of the system under analysis. In other words, the main challenge has

Authors' addresses: F. Pagani and D. Balzarotti, Eurecom, France; emails: {fabio.pagani, davide.balzarotti}@eurecom.fr; O. Fedorov, Igor Sikorsky Kyiv Polytechnic Institute, Ukraine; email: vsnrain@gmail.com.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2019 Association for Computing Machinery.
2471-2566/2019/03-ART9 $15.00
https://doi.org/10.1145/3310355

been to automatically assign individual bytes to the corresponding high-level components, such as running processes, device drivers, and open network connections. This process requires a precise knowledge of the internal data structures used by the target operating system, combined with a set of heuristics to locate and traverse these structures and retrieve the desired information. We call this part the **spatial aspect** of memory analysis, as it deals with the *location* of data objects and with their point-to relationships in the address *space* of the system under analysis.

While this spatial analysis still suffers from a number of open problems [12], memory forensics is a mature field and popular tools in this area are routinely used in a large number of investigations.

In this article, we introduce a second, orthogonal dimension that we believe plays an equally important role in memory analysis: **time**. If the spatial dimension deals with the precise localization of key data structures, the temporal dimension is concerned with the temporal consistency of the information stored in those structures. This new dimension should not be confused with the one introduced by Saltaformaggio [47], where the authors focused on reconstructing a *timeline* of past user activities. Instead, our dimension is tightly related with the memory acquisition process and on how the content of memory changes during this process. In fact, the most common tools that are regularly used to acquire the physical memory of a running machine are executed while the rest of the system is running, and, therefore, while the content of the memory itself is rapidly changing. This is the case of software-based solutions, as well as hardware-based approaches that retrieve memory through Direct Memory Access (DMA). As a result, instead of acquiring a precise snapshot of the memory, these tools obtain some sort of blurred, long-exposure picture of a moving target. This issue is well-known among practitioners in the field. Already in 2005, when memory forensics was still in its infancy, Harlan Carvey posted a message to the Security Incidents mailing list [10] pointing out that the inability of software collection tools to freeze the memory during the acquisition was going to become a problem for future analysis. Unfortunately, this warning remained largely unexplored by researchers for over a decade—while all resources were dedicated to improve the algorithms used to overcome the semantic gap and recover useful information. As a result, existing tools and techniques do not provide any way to estimate, mitigate, or even simply to understand the presence and the possible impact of errors introduced by the lack of atomicity in the collected dumps.

Only recently, researchers have looked at the lack of atomicity in memory dumps, but previous works have focused only on introducing definitions and confirming that, in fact, software acquisitions produce non-atomic memory images [24, 60]. In these studies, the authors reported problems of "inconsistent page tables" in 20% of their dumps, likely due to *page smearing* [12], and resorted to repeat their experiments when this problem was present. In 2018, Le Berre [33] also confirmed the astonishing result that *about every fifth acquisition results in an unusable memory dump*. Sadly, while this is undeniably one of the *"main obstacles to complete memory acquisition"* [12], the forensic community is following a "dump and pray" approach, simply suggesting to collect new snapshots when there is evidence of incorrect results. Unfortunately, this is a luxury that is rarely available in a real investigation.

To mitigate this problem, in this article, we suggest for the first time that the two dimensions (**spatial** and **temporal**) need to be always *recorded* and *analyzed* together. The analysis of the value of memory objects allows the analyst to reconstruct a picture of the state of the target machine and its applications. The analysis of the time those values were acquired provides instead a way to estimate the confidence that the extracted information (and, therefore, the result of the entire analysis) is consistent and correct. Our experiments show that page smearing is just the tip of the iceberg and similar inconsistencies occur everywhere in the memory acquired by live acquisition techniques. In fact, while it is obvious that memory snapshots taken from *inside* a running system are always non-atomic, it is still unclear whether (and how often) this can result in

wrong conclusions during an investigation. To answer this question, we present a number of real examples that we use to pinpoint three main types of time-based inconsistencies in a number of real memory analysis tasks. Our experiments show that inconsistencies due to non-atomic acquisitions are very common and affect several data structures. Moreover, these inconsistencies often lead to potential errors in the analysis process.

Knowing the extent of the problem is important, but it does not help to improve the current memory forensic field. Therefore, we argue that memory acquisition tools should provide the user not only with the raw data required for the analysis, but also with the necessary information to clearly estimate the reliability of this data and the possible impact of non-atomic collection on a given forensic task. To follow this recommendation, in Section 5, we introduce the temporal dimension to two of the most popular memory forensic tools: the Volatility analysis framework and the LiME acquisition tool. Our changes allow, for the first time, to precisely record time information in a memory dump and to transparently support this information during the analysis. Moreover, whenever an atomic collection is not possible, in Section 5.3, we discuss a better algorithm to decrease the impact of the acquisition time on the memory analysis process. In particular, we propose a simple context-sensitive approach to improve the "local-atomicity" of a number of relevant data structures.

2 SPACE AND TIME IN MEMORY ACQUISITION

Tens of different memory acquisition techniques have been proposed to date, all with their combination of strengths and weaknesses. The list includes dedicated hardware solutions [9, 17], DMA acquisition via the FireWire bus [4, 5], and the use of hibernation files [46] or crash dumps [52]. Despite this variety of techniques, software-based acquisition solutions (e.g., `LiME` [58], `WinPMem` [15], and `Memoryze` [39]) remain the tool of choice in any scenario that does not involve emulated or virtual environments—where a consistent snapshot can be acquired from the hypervisor while the underlying Operating System (OS) is frozen.

The first step performed by any kernel-level tool is to retrieve the system memory layout. The layout specifies which memory region are usable by the operating system, and which are reserved for memory-mapped peripherals, such as Peripheral Component Interconnect (PCI) devices, or for the system Basic Input/Output System (BIOS). The only way to retrieve this information is to ask the BIOS itself, which can only be queried during the boot phase, when the system is still in *real mode*. For this reason, OS kernels maintain a copy of this list, which in Linux is pointed by the `iomem_resource` symbol and in Windows is accessible through the `MmGetPhysicalMemoryRanges` API. This step is very important, as any attempt to access reserved regions may result in unpredictable side effects [36] as a device can sense and react to the read requests generated by the acquisition process and place the system in an unstable state (or force it to crash). Once the memory layout has been retrieved, the acquisition process can finally start. For each region assigned to system RAM, the module *maps* a page of physical memory and stores it in a file. In Linux, this is done using the `kmap` kernel API.[1]

For the purpose of this article, we are only interested in two aspects of the acquisition process. The first is whether the target system is running while the content of the memory is copied. In the case of OS-based or hardware-based acquisitions, this is indeed the case. Thus, while the memory dump is collected, new processes are spawned, incoming connections are handled, and files are read and written to the disk. The second aspect is the order in which the individual pages are acquired by the tool. To the best of our knowledge, for efficiency reasons, all solutions acquire the memory sequentially, starting from lower to the higher available page.

[1]Since on several architectures the Linux kernel maintains a direct mapping to the entire physical memory, this API just performs an address translation and does not involve the modification of any paging structure [56].

Once the acquisition process is completed, the real analysis can finally begin. Here, the main challenge is the fact that while a memory dump contains a copy of the *physical* pages, the analyst often needs to reason in terms of OS-provided abstractions, such as processes and their virtual memory space. To bridge this *semantic gap*, memory forensic tools (such as the popular open source *Volatility* [62] and *Rekall* [16] frameworks) require building a *profile* of the operating system that was running on the target host. Such profiles contain information related to relevant kernel data structures and their positions in memory. While the creation of these profiles and their corresponding analysis, what we call the *spatial analysis*, is an active research area (see, for instance, Socala and Coen [55] and Gavit et al. [19, 20]), this article focuses on a different and often overlooked problem: the fact that all existing analysis algorithms are based on the assumption that *all pages were acquired at the same time in a single atomic operation*. Not only is this never the case in the acquisition process we described, but, as we will see in Sections 3 and 4, this can have important consequences on the results of the analysis.

Atomic Acquisition and Time-Consistency. An ideal acquisition procedure would collect the entire memory in a single atomic operation—such that, for the system under analysis, the acquisition appears to be performed instantaneously. For all practical purposes, an acquisition can still be considered atomic if the content of the memory does not change between the beginning and the end of the acquisition process—making the result indistinguishable from a hypothetical snapshot collected in a single operation. Recently, Vomel et Freiling [60] proposed a more permissive definition of Atomicity, inspired from the theory of distributed systems. This atomicity, which we call *Causal Atomicity*, can accommodate pages collected at different points in time, as long as the procedure satisfies the causal relationships between memory operations and inter-process synchronization primitives. While this is a very elegant definition, it is unfortunately also extremely difficult to measure in practice, as it is almost impossible to enumerate all causal relationships in a complex system. In fact, the same authors later estimated the atomicity of real dumps by simply computing the time delta between the first and last acquired pages [24]. Note that this is just an approximation, as even very short time deltas may still result in snapshots that do not satisfy casual relationships.

What matters most is the fact that any kernel-level acquisitions are not atomic (neither according to the original nor the causal definition) [24]. To better examine the consequence of this fact, we need a more fine-grained measure of the discrepancy between different parts of a memory dump. For this reason, we introduce the concept of *time consistency*. A set of physical pages are *time-consistent* if there exists a hypothetical atomic acquisition process that could have returned the same result or, in other words, if there was a point in time during the acquisition process in which the content of those pages co-existed in the memory of the system.

We argue that time-consistency is a more practical measure of the quality of a memory snapshot, as it can be accurately measured in an experiment (as we will describe in Section 4). Moreover, while it is still practically impossible to obtain a complete time-consistent image if the system is not frozen during the acquisition process, it is still useful to test this property on a smaller scale. For instance, an analyst may be interested in knowing if all the `EProcess` structures contained in a memory dump are time-consistent. In other words, even if the entire memory dump contains non-consistent data, individual processes or particular data structures may locally satisfy this property. Therefore, if we know that the acquisition of the process list is time-consistent, this may be enough to guarantee the correctness of a number of important memory forensic tasks.

3 IMPACT OF TIME IN MEMORY FORENSICS

The memory of a running system can be seen as a very large graph of interconnected objects. This is essentially what makes memory forensics possible in the first place. However, as we explained

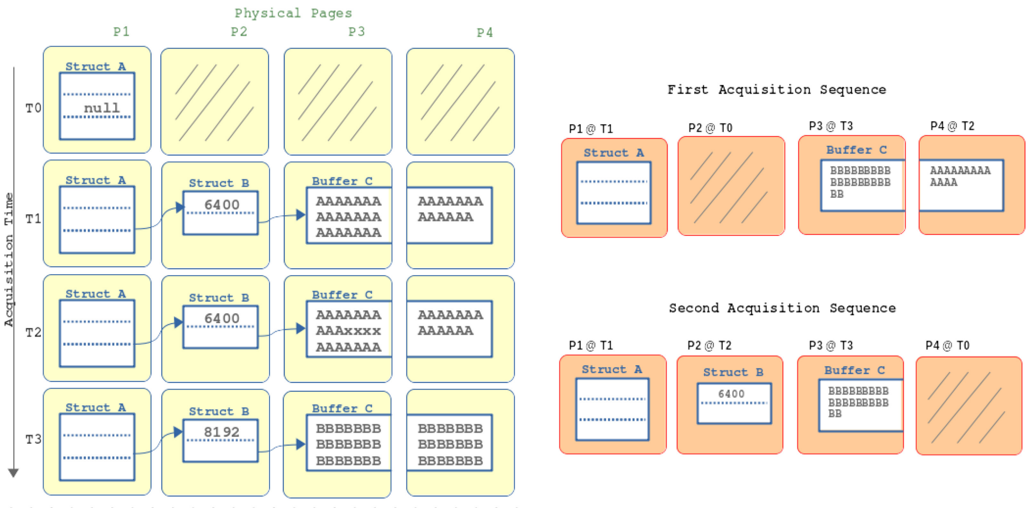

Fig. 1. Example of non-atomic acquisition.

in the previous section, those objects are often collected at different points in time—with the result that the values stored in different parts of the memory can be inconsistent and that pointers (i.e., the edges in the graph) can be incorrect and point to the wrong destination.

Figure 1 shows an example of the problems that can occur during a non-atomic acquisition. The left of the figure shows (in a simplified manner) three distinct but interconnected memory objects: a `Structure A`, which contains a pointer to a `Structure B`, which in turn points to a `Buffer C`. The two structures are stored in two separate physical pages (P1 and P2) while the buffer spans two pages (P3 and P4). All pages are represented sequentially along the X axis of the figure. The Y axis shows instead how the memory content evolved over time (represented in four discrete points: T_0, T_1, T_2, and T_3). As an example, we can imagine that at Time T_0, only `Struct A` is allocated. The other two objects are created at Time 1, when the buffer is also filled with 6,400 characters (as indicated in the first field of `Struct B`). Part of the buffer is then modified at Time T_2, and finally, its entire content is replaced by 8K new characters at Time T_3.

The right part of the figure shows two possible acquisition sequences for the four physical pages. We use the notation `Px@Ty` to indicate that Page X is acquired at time Y. For instance, the first dump corresponds to the acquisition sequence: $P_1@T_1$, $P_2@T_0$, $P_3@T_3$, $P_4@T_2$. We now use these two examples to introduce three types of inconsistencies that can affect a memory dump.

Fragment Inconsistency—This problem affects large objects that are fragmented over multiple physical pages, when their content (as acquired in the memory dump) is not time-consistent. For example, in the first acquisition in Figure 1, the buffer contains half of the content it contained at time T_3 and half of the one at time T_2. If this was data received over the network, such as a web page, a forensic investigation would find a mix of two consecutive messages, or a mix of the HTML code of two separate pages. While, in our example, we show a case of fragment inconsistency on a buffer, this can also affect large structures or arrays containing multiple elements.

Pointer Inconsistency—The second type of inconsistency affects the connection between two different objects, when the value of a pointer and the data it points to are acquired at different moments in time. This is the case for `Struct A` in the first acquisition sequence. Its pointer to `Struct B` was acquired at time T2, while the content of the destination page was acquired at T1, before the second structure was even allocated. This can have serious and unpredictable effects on

memory analysis. In fact, a forensic tool may try to follow the pointer and cast the destination bytes to a `Struct B` type. Sometimes the result can be easily identified as incorrect, but unfortunately, arbitrary sequences of bytes can often be interpreted as valid data, thus leading to wrong results.

Value Inconsistency—The third and last type of inconsistency we present in this article occurs when the value of one object is not consistent with the content of one or more other objects. This is the case of P2 and P3 in the second acquisition sequence of Figure 1. The pointer in `Struct B` is not affected by the non-atomic acquisition, and it correctly points to `Buffer C`. However, the counter in the structure that keeps track of the number of characters currently stored in the buffer is not consistent with the content of the buffer itself (when its value was 6,400, the buffer did not contain any "`B`"). The page smearing phenomenon recently described by Case and Richard [12] is also an example of value inconsistency—in this case, between the data stored in the page tables and the content of the corresponding physical pages.

As we already mentioned in the previous section, the fact that two pages are collected at a different point in time does not necessarily mean their content is inconsistent. For instance, P1 and P2 in the second dump of Figure 1 are time-consistent. Also, pages P1 and P3 are time-consistent. However, all three of them together (P1, P2, and P3) are not.

4 IMPACT ESTIMATION

In this section, we evaluate how often the inconsistencies presented in the previous section are present in a memory dump and what are the consequences on a number of common memory analysis tasks.

All experiments were conducted using a virtual machine equipped with 4 CPUs, 8GB of RAM, and running Ubuntu 16.04 LTS. While the problem of non-atomic dumps is independent from the operating system, we decided to base our tests on Linux because the availability of its source code simplified the task of retrieving and double-checking the internal state of kernel objects. The physical memory was acquired using LiME [58], a popular kernel-level acquisition module for Linux and Android systems.

We want to stress that while our experiments were conducted in a test environment, we strongly believe that this does not invalidate our findings. Moreover, the memory acquisition speed of our environment is comparable with the one exhibited by the fastest tool, as reported by McDown et al. [42]. This puts our experiments in a best-case scenario, and thus, we believe the use of slower tools can increase the scope and the number of inconsistencies present in a non-atomic memory dump.

4.1 Fragmentation

In our first experiment, we want to show how *fragmented* the physical address space is in an average computer. In fact, consecutive pages in the virtual address space of a process may be mapped to very distant pages in the system RAM. The actual location may depend on many factors, including the OS allocation policy, the number of running processes, the amount of available memory, and the uptime of the system (memory in a freshly booted computer is likely to be less fragmented than memory on a server that has not been restarted for months). For our test, we took a conservative approach, using a Linux VM that had been running traditional desktop software (e.g., Firefox and VLC media player) for a period of only 3 hours. We expect a dump of a real system to be even more fragmented than what we measured in our test.

Figure 2 shows the physical pages assigned to a subset of the processes running in the system. The X axis shows the position (all pages were acquired sequentially) of the physical pages—where blue squares represent program code and data, green circles the process stack, and red triangles

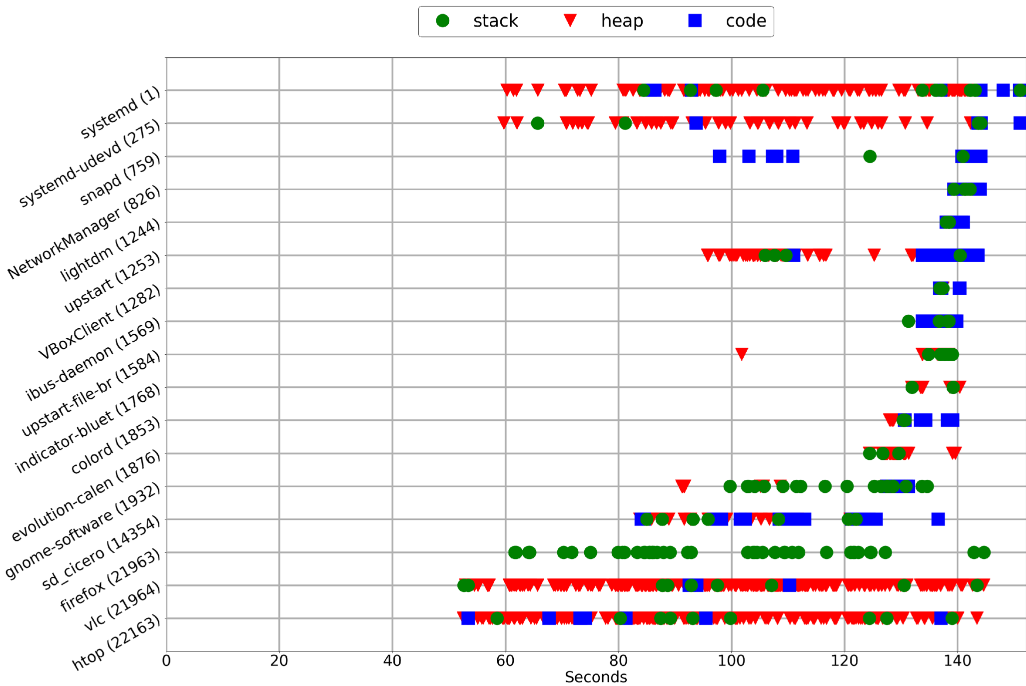

Fig. 2. Example of physical memory fragmentation.

heap pages. Since the tool acquired each page in the order in which they appear, the X axis also measures the time at which each page was acquired.

Thanks to this figure, we can observe several interesting phenomena. First, for some processes (such as NetworkManager), all pages were located close to each other in the physical memory. Therefore, these pages were also acquired in a very short period of time. For other processes (e.g., snapd and udevd), the pages were instead scattered through the entire RAM, thus increasing their inter-acquisition time and the probability of containing inconsistent data. Another interesting case is given by the Firefox process, for which our analysis was unable to locate the physical pages containing the program code section. A closer analysis revealed that this was due to an *inconsistency* in the process VMA list (which, as we explain later in this section, Linux uses to store the memory region owned by a process).

To investigate this type of issue and better understand how the fragmentation may affect the results of an analysis, we performed two sets of experiments—one focusing on the integrity of kernel data structures, and one focusing on similar issues in the address space of a single process.

4.2 Kernel-Space Integrity

The OS kernel contains many useful pieces of information that are required during a forensics investigation. This information is spread over a multitude of interconnected data structures that evolve over time to keep track, for example, of resource allocation and the creation of new processes. These structures are connected by means of pointers in complex topologies such as linked lists, red-black trees, and graphs. A classic example of such topologies is the process list. In the Linux kernel, every process is represented by a task_struct, which plays a central role in the kernel as it stores most of the information associated to each running process. At the time of writing, this structure contains more than 300 fields, and 90 of them are pointers to other structures. One

```
1  struct mm_struct {
2    /* list of VMAs */
3    struct vm_area_struct *mmap;
4    /* RB tree of VMAs */
5    struct rb_root mm_rb;
6    ...
7    /* number of VMAs */
8    int map_count;
9    ...
10 } ;
```

Fig. 3. An excerpt of `mm_struct` taken from `linux/mm_types.h`.

of these fields, called `tasks`, is particularly important because it is by following the pointers contained therein that a tool can enumerate the active processes in the system. Another valuable field part of the `tast_struct` is `mm`, which points to a structure (`mm_struct`) that contains all the information related to the virtual memory of the process, such as the first and last address of the stack and the heap, and a reference to every memory mapping requested by the userspace program.

Because of its central role in inspecting the memory of a process, we focused our evaluation on this structure. Page tables could have been another possible candidate for this test, as other researchers already noted inconsistencies in the page table when they are acquired non-atomically [12, 24]. However, while this is known, the impact on other kernel data structures is still, as of today, unclear. Moreover, all modern operating systems implement *demand paging*, which means that user space pages are not loaded and mapped in RAM until they are needed—thus complicating the task of identifying possible inconsistencies. While this mechanism can be disabled by an application (i.e, using the `mlock` Linux system call), this would provide an unrealistic scenario on which to carry out our test.

Figure 3 reports an excerpt of the `mm_struct` definition. The first important field is the `mmap` pointer, which points to a `vm_area_struct` (VMA). Each of these structures represent a different memory region (such as a shared library, the code of the program, or its stack) and contain the information about the beginning and the end of a map, the permissions associated with it and, in case there is one, a pointer to the mapped file. This is the information that is used to populate the `maps` file under the `proc` filesystem.

A memory forensic analysis tool can reach these structures in two different ways. The first one is by traversing a linked list containing all `vm_area_struct`. The second is by traversing a red-black tree, which contains the same elements of the linked list and is rooted in the `mm_struct`'s `mm_rb` field. This redundant design allows the Linux kernel to search for a free area using the list, but also to take advantage of the tree topology to quickly check—for example, during a page fault—if an address belongs or not to a VMA [22]. The last field that is important to understand for our evaluation is `map_count`, a counter that contains the number of VMAs associated to the process.[2]

We now want to understand what happens to these data structures when a memory dump is collected in a non-atomic fashion. In fact, inconsistencies in these structures can result in serious problems during an investigation, as it would be impossible to know where the memory containing the program code, the shared libraries, the heap, and the stack of a process is located. However, it is important to note that all the structures we mentioned so far are allocated using the `kmalloc` function—which ensures that two contiguous pages in the virtual address space are *also* contiguous

[2]While these structures are specific to the Linux kernel, an equivalent tree is also present in the Windows operating system, under the name of Virtual Address Descriptor (VAD) tree. The use of the VAD tree in memory analysis has been previously described by Dolan-Gavitt [18].

Table 1. Experiment Results

	Scenario 1 (Firefox)	Scenario 2 (Apache)	Scenario 3 (Malware)
List mismatch	100%	71%	80%
Tree mismatch	100%	73%	80%
Total	100%	78%	80%

in the physical memory. For this reason, even large structures cannot be affected by the *Fragment Inconsistency* problem we described in the previous section.

Experiments. To mimic different real-world scenarios, we performed three experiments. In the first one, we re-created a scenario where the investigator needs to analyze the memory of a Firefox web browser that was left with five open tabs. The second scenario involves, instead, a potentially infected server machine, hosting a Wordpress installation, and handling a workload of 15 requests per second. In this case, the examiner wants to analyze the content of the Apache memory, to understand if the server was compromised. The last scenario involves, again, a client machine, this time infected by a real malware.

For each scenario, we collected 10 different memory dumps, which were then analyzed by a custom Volatility plug-in designed to look for inconsistencies among the VMA-related data structures. In particular, for each process under analysis,[3] it traverses both the list and the tree of VMAs, *counting* the number of elements they contain. It then compares these two numbers with the `map_count` field and prints an error message if the two values are different. The list exploration algorithm implemented in Volatility was left untouched. On the other hand, we had to fix the recursive exploration of the tree since the Volatility implementation was hanging on some memory dumps. We believe this is due to the presence of malformed trees (a consequence of the non-atomic acquisition), for which the exploration was trapped inside an infinite loop.

The results for all the three scenarios are presented in Table 1. The first row of the table reports the fraction of processes for which the number of elements contained in the VMAs list and the `map_count` counter were inconsistent. Similarly, the second row contains the result of doing the same comparison, this time counting the number of elements in the red-black tree. The last row shows how many processes are affected by one or the other inconsistency. A stunning 78% to 100% of the analyzed processes in the three scenarios contained errors in their VMA information.

It is important to understand that a mismatch between the counter and the elements in the list (or in the tree) does not necessarily translates into a serious problem or into wrong results, but it should nevertheless alarm the analyst. It is indeed possible that after the counter was acquired, the process created a new memory mapping, thus increasing the number of elements in the data structures *without* seriously affecting the analysis of the process memory.

We, therefore, took a closer look at the elements present in those data structures, searching for some mappings that should always been present: namely, the application *code* and its *stack* (we did not include the *heap*, as programs like Firefox use their own custom dynamic allocator and, therefore, there is no corresponding entry in the VMAs structures). Surprisingly, in the Firefox experiment, the list of mappings *never* contained any entry for the application code nor for its stack. Being compiled as position-independent code, the program is loaded into a fairly high part

[3]In the malware and browser scenarios, we analyzed only one process per memory dump. However, Apache in its default process management mode spawns several processes to simultaneously handle all the incoming connections. Therefore, over the 10 dumps, our tool analyzed the status of 153 `apache2` processes. For this reason, all results are reported as percentages.

of the virtual address space. The VMA list is sorted according to the starting address of the memory area, and Firefox had over 200 mapped entries that preceded the code. Therefore, any change in those entries can result in a corruption of the list that prevents the discovery of the area where the code is located. A closer look revealed that these mappings contained the custom heap of Firefox, which changes very rapidly—thus increasing the probability of an error in a non-atomic acquisition. The picture was only slightly better for the red-black tree, where the code mapping was found 3 out of 10 times, and the stack only once.

A similar result was found in the malware experiment. In this case, the code and stack sections were missing in five out of eight inconsistent dumps according to the red-black tree. Using the list, it was instead always possible to retrieve the code entry, while the stack was missing in three out of eight experiments.

Atomicity and Quiescence. Even a perfectly atomic dump acquired on a frozen OS can contain errors in some data structures, if the acquisition was performed while the operating system code was in the middle of a data structure update [29] (e.g., while adding or removing an element from a list). Therefore, one may argue if the inconsistencies we detected in our experiments are due to the lack of time-consistency or to the fact that the image was taken while the OS was not in a quiescence (i.e., idle) state.

While it is not possible to distinguish the two cases, we are confident that time was the culprit of the errors we detected for two important reasons. First, because while the OS can indeed be in the middle of updating some pointers, this can only affect a very limited group of data structures in a dump (i.e., the OS cannot be in the process of updating the memory maps of *all* running processes). Second, when we repeat the same three experiments using a different acquisition strategy that can preserve the atomicity of certain memory regions (see Section 5 for more details), all the inconsistencies we reported in the previous experiments disappear.

Impact. To conclude, we want to emphasize why this seemingly insignificant problem can have a serious impact on an investigation. First of all, inspecting the memory of a malicious process, or of a potentially compromised application, is a common task in memory forensics. If the analyst cannot trust the VMA information, important areas of memory can be missed or wrongly attributed to a different process. On a different example, a few days after the spread of the `WannaCry` ransomware attack, a decryption tool (`wannakey` [26]) was published by a security researcher. The tool works by extracting from the malware memory the prime number of the private `RSA` key used by the ransomware to encrypt the victim files. In a similar attempt, a Volatility plug-in [40] searches for the `AES` keys used by the `NotPetya` malware. This shows how memory analysis often relies on locating and extracting a small amount of data from a process memory. But our experiment highlights that the apparently trivial task of retrieving all memory regions mapped by a process results in wrong information in *over* 80% of the dumps acquired with a non-atomic solution.

4.3 Userspace Integrity

We now shift our focus to userspace applications, to understand if they are also impacted by similar problems. Most of the data from a running application can be retrieved by looking at its stack and heap segments. The heap contains dynamically allocated objects, which are often aggregated in complex high-level structures such as lists and trees. Therefore, the type of errors introduced by inconsistent pages is similar to what we already discussed in the kernel-level experiments. On the other hand, the stack has a special role in the process execution, as it is responsible for maintaining information about the current state of nested functions invocations, along with the values of their parameters and local variables. This is very important to understand what the program was doing at the time the memory was acquired, and to extract the pointers to dynamic objects required to

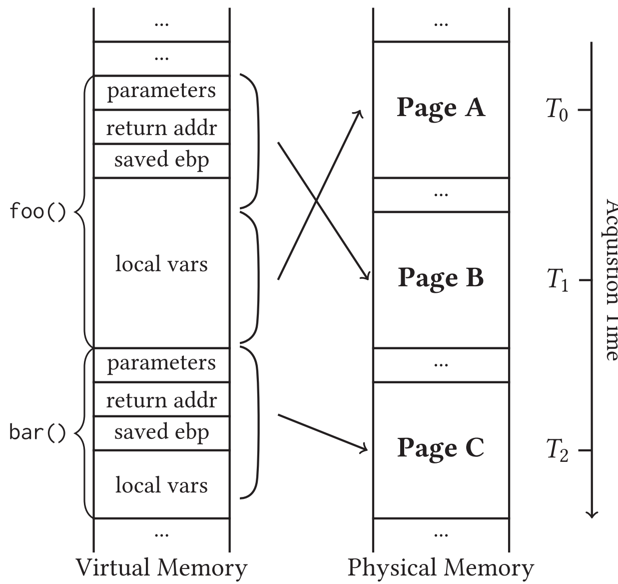

Fig. 4. Stack layout of the target program. On the left, a view from the virtual memory perspective, on the right, from the physical one.

initiate a heap analysis. For this reason, in our second set of experiments, we decided to look at the consequences of non-atomic dumps on the analysis of the stack of a program.

To perform these tests, we need to be able to reliably reconstruct the stack trace of a process starting from its memory dump. This task may be complicated by a number of different factors (e.g., library functions that do not use the frame pointer, or lack of symbols information). However, since our goal is to investigate inconsistencies in the stack pages, without loss of generality, we can prepare the environment to facilitate this analysis. In particular, we compiled the target application (`bzip2`) and all its libraries without any optimization (`-O0`). We then disabled the stripping of debug information, to retain the function names and corresponding addresses in the binary.

Figure 4 shows a portion of a stack trace. The initial values of `rip` and `rbp` are obtained from the *kernel stack* of the process, reachable from the process's `task_struct`. This is where the kernel saves the content of all registers upon a context switch. Using this information, we can locate the current function and then walk back to the previous frames by following the `saved rbp` field. The left side of the figure shows the activation records of two hypothetical functions (`foo()` and `bar()`). It is important to note that while the two frames are always contiguous in the virtual memory of the process, the physical pages that contain them can be quite far apart, and even appear in a different order (right side of Figure 4). Moreover, the frame of `foo()` in our example spans across two separate memory pages, thus fragmenting its content in non-adjacent physical addresses. Finally, the figure also shows the temporal axis that marks when each part of the memory was collected by a hypothetical acquisition process. In our case, the first part of `foo()` activation record was dumped at time T_1 and its second half at time T_0.

All three types of inconsistency introduced in Section 3 are potentially present in our example: fragment inconsistency in the local variables of `foo()`, value inconsistency among the content of the two frames, and pointer inconsistency between the `saved rbp` of `bar()` and the content of `Page B`. Moreover, the kernel stack of the process—which is where the initial value of `rbp` and `rip` are extracted from—also lives in another memory page, which could potentially be collected long

Table 2. Experiment Results

	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8	D_9	D_{10}
Frames	—	6	—	6	8	—	—	—	6	—
Physical Pages	4	5	5	4	4	5	4	4	5	5
Acquisition Time (s)	3.2	30.0	37.8	37.0	0.25	26.0	28.6	1.0	27.6	39.9
`rbp` delta (s)	7.7	38.8	49.6	43.7	7.3	43.4	4.3	4.0	15.1	5.64
Corrupted (`registers`)	✓	—	✓	—	—	✓	✓	✓	—	✓
Corrupted (`frame pointers`)	—	—	—	—	—	—	✓	—	—	—
Inconsistent data	N/A	✓	N/A	✓	—	N/A	N/A	N/A	✓	N/A

after or long before T_2. In this section we will therefore try to measure how the lack of atomicity in the acquisition of the userspace and kernelspace stacks affects the process of creating a correct backtrace of a running application.

Experiments. To identify which errors are introduced by the non-atomic acquisition we need a ground truth to compare with. We solve this problem by modifying the LiME tool to keep track of changes on the stack content during the acquisition time. In particular, we modified LiME to save an atomic view of the target process' stack *anytime* one stack page was acquired by the normal acquisition process. In our example, this resulted in acquiring the content of the three pages (`Page A`, `B`, and `C`) at T_0, T_1, and T_2. The atomic view also included the value of `rbp` and `rip` taken from the kernel stack. To rule out the possibility that our kernel module and the target program run at the same time on two different CPUs, therefore invalidating the registers saved in the kernel stack, we executed this experiment using a VM with a single CPU. Also, to ensure our kernel module does not get preempted by other code with higher priority while is acquiring the atomic view, we used the `lowlatency` version of the Ubuntu kernel, which is compiled with the `CONFIG_PREEMPT` kernel option. This enables two additional macros that allow our code to disable preemption before acquiring the required elements and to re-enable it afterward. These macros are enough to protect a small critical section, but our module calls complex kernel functions that may "intentionally" call the `schedule()` function to pass control to another task. When this happens, the kernel produces a warning message that we can monitor to discard the image and repeat the test until no warning message is generated.

This complex setup was required to be able to reliably monitor what happens to the process stack during the memory acquisition process. Therefore, we can expect that in a real setting with multiple CPU cores executing concurrent code, the number of inconsistencies would be substantially higher.

We repeated the experiment 20 times, 10 on an idle system and 10 on a system under a high workload. Since we did not find any significant difference among the two sets, we present in Table 2 only the result for the idle system. Each column in the table represents a different memory dump. The first four rows show some statistics about the stack trace of the target process, including the number of identified frames and the number of stack pages effectively used by the application. The third row reports the time difference (in seconds) between the first and the last acquired page of the stack, while the fourth row contains the difference between the kernel stack acquisition time and the time the page containing the top stack frame was acquired. Intuitively, the larger the two time windows, the more likely the process was to modify the stack during the acquisition, resulting in possible incorrect results for the analysis.

The bottom half of Table 2 reports the problems we run into when running our Volatility plug-in to extract the stack trace on each memory dump. The table lists three separate cases. In the

first two, marked as *Corrupted*, our plug-in was unable to generate a complete stack trace. This was due to two different reasons. The first (marked as `registers` in the table) is an inconsistency between the registers stored in the kernel stack and the state of application stack, which caused `rbp` to point to an incorrect location. This was the case for 6 out of 10 dumps. However, one may argue that an analyst may resort to other heuristics to locate the top frame on the stack, without the need to recover the registers from the kernel. Therefore, we re-run our Volatility plug-in by providing the initial correct value for `rbp`, and, therefore, the position of the top frame, as a parameter. The next row in the table (marked as `frame pointer`) reports the cases in which, despite the top frame being identified manually, it was still impossible to retrieve the entire stack trace. In this case, the problem was that adjacent stack frames resided in *non-adjacent* physical pages acquired at different points in time. As a consequence, the saved `rbp` of one function pointed to a page that had been already overwritten with other data.

For the four dumps for which it was possible to reconstruct the stack trace, the last line in Table 2 shows the images in which at least one stack frame spanned multiple physical pages containing inconsistent information. This happened in three out of four cases (marked as `Inconsistent data` in the result table).

To conclude, only in 1 out of 10 dumps was it possible to retrieve a correct backtrace that was neither corrupted nor inconsistent. In other words, it seems that errors introduced by non-atomic dumps *are the rule and not the exception*. Not surprisingly, the only correct case corresponded to the acquisition in which all the stack pages were collected in less than 250 milliseconds. This, as we better explain in Section 5, prompted us to look for possible solutions to minimize the acquisition time windows.

Impact. Researchers have recently proposed several forensic analysis techniques tailored for user space applications [3, 7, 13, 38, 44]. Despite this effort, forensic tools still support only a *"small percentage of applications that hold forensic value"* [12] and experts are thus urging the community to extend existing tools to support web browsers, office applications, and web and database servers. While the methodology used to locate and extract information from these applications may be very different, they all share a common unwritten assumption: they *require* a consistent view over the data of a process, whether it is stored on the stack or on the heap. We believe that our experiments draw attention to a problem too often overlooked by the forensic community, and will help researchers to better assess their methodologies in the presence of non-atomic acquisitions.

5 A NEW TEMPORAL DIMENSION

The importance of the temporal dimension has been largely underestimated in memory forensics. In the previous sections, we showed that this negligence can lead to wrong results during several memory analysis tasks. While our experiments shed light on an important problem, we did not discuss possible ways to mitigate the problem.

In this section, we tackle the problem by following two different approaches. First, we discuss how we can record precise timing information during the memory acquisition phase and integrate this information in the forensic analysis process. Second, we discuss how the lack of atomicity can be mitigated, by performing a non-linear acquisition scheme.

5.1 Recording Time

Given an object and its address in memory, there should be a way for the analyst to know the exact moment in time in which that part of the memory was acquired. This gives the forensics analyst the ability to estimate the degree of atomicity among two or more objects used during an analysis task. To record this information in the first place, we modified LiME [58] to produce, along with

the memory content, a log file containing precise time information about the acquisition process. Our implementation allows the user to save a timestamp for each page, or to only output the time information at particular intervals (e.g., once every 10ms).

We then conducted three experiments on a bare metal machine equipped with 8GB of RAM: a baseline test without any time acquisition, one test that saved a timestamp every $100\mu s$, and one that logged the actual acquisition time for every page. The baseline acquisition was completed in 83.92 seconds. The second experiment logged over 85,000 timestamps with an almost negligible overhead of 0.6 seconds (0.7%). Finally, the third experiment logged over 2M timestamps with 1.98 seconds (2.4%) overhead. Since any extra time spent to record the dump increases the likelihood that kernel and application data were modified during the process, our approach let the user choose the balance between the extra acquisition time and the precision of the logged timestamps.

At this point, one may argue that a much simpler and more efficient solution exists to the time recording problem. It is in fact possible to simply take the total time spent to acquire the entire memory, assume it was uniformly distributed among all pages, and divide it by the number of physical pages to compute the time required to dump a single page. Given this number, it is trivial to label each physical page with an *estimation* of its acquisition time. This solution is appealing at first glance, as it does not require changing existing tools and does not introduce any overhead in the acquisition process. However, we observed that, in reality, the acquisition code does not always run at the same pace from the beginning to the end of the dump. This is due to variables related to the state of the system that are impossible to evaluate *a posteriori*, such as the status of disk buffers and caches, or the fact that other programs may suddenly require more CPU or may start writing to the disk, thus affecting the acquisition speed. To verify what is the error introduced by using this naive approach, we compared the estimated timestamp with the real one collected using our version of LiME configured to log the time for every page. Obviously, the *skew* between the timestamps of the first and last page is zero. For the remaining pages, the average skew was 4.4 seconds, with a maximum error of 8 seconds—which is unfortunately way too high to properly perform any time-based analysis. The modified LiME software can instead achieve very high precision (less than 1ms error) with less than 1s overhead to acquire 8GB of RAM—which we believe is perfect to be integrated into production systems.

5.2 Time Analysis

We now look at how we can integrate in Volatility the time information we recorded. Our goal is to add this new dimension transparently, in order to be able to enrich the output of every command and plug-in that is already available for the framework.

Volatility internally represents every OS structure with a dedicated standalone object, which contains different information such as its name, the data type of its fields, and the memory address where the structure is allocated. We modified the constructor of this object's class to retrieve and store the timestamp associated to the physical pages where each structure is stored. As Volatility reads some memory regions without passing through one of the structure objects (for example, when it walks the page table), we also patched the code responsible to handle direct reads. All timestamps are then recorded inside an *access timeline*, which is printed, along with other statistical information, after any Volatility command is completed. Moreover, the timing API we developed are exported to plug-in developers, so other code can easily perform queries to know when a particular structure was acquired. Our current prototype presents to the investigator the number of physical pages accessed by the analysis, the time window in which those pages were originally acquired, and a simple graphic representation of the *access timeline*. A new parameter, `pagetime`, enables the timeline tracking and the display of this summary.

Table 3. Selected Subset of Volatility Plug-ins

Name	Description	Acquisition Timeline Bar
bash	Recovers bash history	[xX--------------------------XXXXXXX----XXX---X--X---XX--X-XX---XX-XXXXXXXX-XXXX-]
dmesg	Gathers dmesg buffer	[X---]
dump_map	Dumps mappings to disk	[XX------------------------XxXXXXXxX---xXXX--xX-xX-xxxXxxxxxX--xxX-XXXXXXXxxXXXXX]
elfs	Finds ELF binaries	[XX------------------------------X-----XXXX---X--X----X-X--------X-X-XXXXX---XXXX]
ifconfig	Gathers active interfaces	[X---X----XX-]
info_regs	Prints kernel stack registers	[xX--------------------------------------x-------X---------------X-X-XXXXX----XX-]
lsmod	Gathers loaded kernel modules	[x---X--------X--]
lsof	Lists file descriptors	[XX-----------------------------------X----------X------------X--X-X-XXXXX----XX-]
proc_maps	Gathers process memory maps	[xX------------------------------X---------------X---------------X-X-XXXXX----XX-]
psaux	Gathers process command line	[xX---------------------x---xxxxxXxXxxXxXXxxxx-x-Xxx-XxxxxxxxxxxXxXxXXXXXXXxXXXx]

Here is an example of the output for the pslist plug-in:

```
$ ./vol.py -f dump.raw --profile=... --pagetime pslist
<original pslist output>

Accessed physical pages: 171
Acquisition time window: 72s
[XX-------------XxX---xXXX--xX-xX---Xxx-xx-X-XxxX-XXX]
```

This information provides a first insight into the level of atomicity of the data structures used during a given task. If they are very dispersed over the memory, it means that Volatility traversed structures that were taken far away in term of acquisition time. On the other hand, a narrow acquisition time window means that the information required to run the task is less likely to contain inconsistencies, and the final result is therefore more reliable.

As an example, Table 3 shows a subset of common Volatility plug-ins, alongside a short description and the barplot of the physical pages used by the command when executed on our 8GB test machine.[4] We selected these particular commands as they traverse a different set of data structures, related to processes, files, network, and memory management. As expected, the outcome of this experiment is quite worrying. Only 1 plug-in out of 10 (dmesg) is using structures collected close in time. The rest rely on structures spread over the entire memory and therefore collected tens of seconds apart.

Note that our timeline can only *suggest* the presence of inconsistencies, pointing the analyst to result that might be incorrect and that therefore should require extra validation. Unfortunately, it is not possible to detect the *actual presence* of inconsistencies, since Volatility has only access to one dump and has therefore no ground-truth information to compare it with. However, it is possible to modify individual plug-ins to detect and report particular cases of *Value Inconsistencies*, such as the difference between the counter of VMAs associated to a process and the number of entries in the VMA list discussed in Section 4. This information could be used to print more fine-grained warning messages to further alert the analyst that particular care must be put in validating the obtained results. However, this process would require manually changing each individual plug-in based on the actual semantics of the information it processes, which is outside the scope of our article. Nevertheless, the presence of our temporal API allows plug-in developers to move in this direction.

[4]Note that all bar plots have an x in the first position. This is due to an access at the beginning of the physical memory that contains the data section of the kernel. This section contains the init_task variable that is used by almost every plug-in as an entry point for the process list.

5.3 Locality-Based Acquisition

So far, we discussed how we can enrich a memory dump with timing information and how this information can be integrated in a memory forensics analysis tool. We now investigate if it is possible to mitigate the side effects of the lack of atomicity and reduce the serious inconsistencies depicted in Section 4.

As already explained in Section 2, the normal approach adopted by LiME and other memory acquisition tools is to scan the memory from the lowest up to the highest physical address assigned to system RAM and save the content of each page *sequentially*. This simple approach treats each memory page as equally important, whether or not it is actually used by the system and independently from the fact that it may contain crucial forensic information.

We propose instead a different approach that maximizes the "local atomicity" by acquiring the memory in consecutive chunks based not on their position but on their content. The idea is that pages that contain interconnected data (e.g., the element of a linked list) should be acquired in a short amount of time to minimize the chance of errors. Our acquisition algorithm is divided in two phases. The first, which we dubbed as *smart dump*, uses OS information to locate and dump a number of physical pages. For instance, it groups together pages that host popular forensic-related information such as the list of processes and the list of loaded kernel modules. Moreover, for each process, it dumps its page table, the `cred` structure (which contains the security context of the process such as the `UID` and the `GID`), the list of memory mappings and the content of the heap and stack segments, the list of open files and the related structures, and the kernel stack of the process that contains the value of its registers after the last context switch. This ensures that while two different applications can be acquired far apart in time, the data of a single application is dumped in the minimum amount of time possible.

The second part of the acquisition consist of the traditional sequential dump of the remaining physical pages. To avoid the overhead of acquiring the same page twice, the smart dump keeps track of the acquired pages in a bitmap. In this way, no single page of memory is acquired more than once. Our experiments show that the additional logic in the smart dump and the bitmap management add a *negligible* overhead to the acquisition time. Moreover, the kernel module is only 15 kilobytes larger than the one used by the traditional linear version. On the memory footprint side, the biggest object allocated by our module is the bitmap. For 8GB of RAM, it requires 256KB of memory, which are nevertheless allocated in an area reserved only for kernel modules. Moreover, given that our solution does not need to allocate specific kernel structures, it does not have any impact on kernel caches—which often contain valuable information [11].

Note that our acquisition sequence relies on information provided by the OS, which are under control of the attacker in a compromised system. For instance, a rootkit can hide an entire process, which, therefore, would not be acquired during the smart dump phase. But even in a compromised system, our smart dump does not produce worse results compared with a traditional linear acquisition. In fact, since all the other processes are acquired in a local-sensitive way, this forces the remaining pages (including the hypothetical malicious process) to be also acquired in a shorter time window during the linear phase.

The same effect is obtained if a malware tries to intentionally increase its memory footprint to force our acquisition algorithm to spend more time acquiring useless pages. Again, this would force the system to postpone the acquisition of other kernel data structures in the remaining time (as the total acquisition time is constant and cannot be manipulated by a malicious actor)—thus resulting in a more consistent dump overall. The only successful strategy would be to increase the memory footprint and position key pointers far away from their targets. This would result in possible inconsistencies using our solution, but certainly not worse than what would be experienced with existing approaches.

Finally, we want to stress that our locality-based acquisition does not require any additional information that is not already used by linear-based approaches—i.e., the kernel headers.

To test if this solution would be sufficient to avoid the problems we detected in the experiments presented in Section 4, we re-run all user- and kernel-space integrity tests on memory dumps acquired using our locality-based approach. In *all* cases, we were able to retrieve a correct stack trace, thanks to the fact that the physical pages containing the process and kernel stacks were all acquired in less than 40 milliseconds. Similarly, the VMAs list and tree structures were also collected together, resulting in no inconsistencies and no missing mapped sections. These results apply to any other analysis that relies on information extracted from the stack or heap of a process, or on the content of the kernel data structure currently supported by our smart dump phase.

6 DISCUSSION

While the analysis and the results presented in this article are limited to few selected data structures, it would be extremely difficult to do otherwise. In fact, extending the test to all data structures is infeasible as a running Linux kernel includes thousands of different structure types. Moreover, to detect the presence of a corrupted information, we need to compare against the same information taken from a different source. For instance, in one of our examples, we compare the VMA counter, the number of elements present in the list, and the one in the red-black tree. Unfortunately, this "ground truth" is not always available and ad hoc rules and experiments would be needed to support each individual structure. However, since there is nothing special about the VMA structure and since the lack of atomicity affects equally all memory, there is no reason to believe that inconsistencies would not affect other data structures managed by the kernel. We actually believe that our experiments were very conservative, and a real running system would suffer even more from this problem.

It is important to stress the fact that the goal of this article is not solve the problem of lack of atomicity in memory dumps. It is instead to draw attention to this important and overlooked problem and show that it is not simply something we need to accept as a factor out of our control. Our contribution is to provide the tools to make the lack of atomicity evident to the analyst so that *time* becomes an aspect that can be measured and taken into account in an analysis. Second, we show that a linear acquisition, as implemented by all tools on the market, is not an optimal solution and better approaches can be easily implemented to reduce errors due to the passing of time. Therefore, more research and more papers are certainly needed to improve the forensic field, not just from its *space* but also from its *time* dimension.

7 RELATED WORK

The forensics community has focused on many different problems over the last decade, including the application of live forensics [1, 31, 34], the creation of signatures for kernel data structures [20, 21, 37], the evaluation of different acquisition methods [59, 61], the de-randomization of the kernel memory [25], the extraction of hypervisor-related information [23], and the design of a number of different analysis techniques tailored to common user space applications [2, 3, 7, 13, 38, 44, 53, 54].

Even though the lack of atomicity is mentioned by several studies [28, 32, 35, 43], only very few works have focused on this topic. The first effort to evaluate the consistency of a memory image was done by Huebner in 2007 [30]. This work acknowledges that capturing a memory image from a live system can indeed introduce inconsistencies between interconnected objects. To resolve this issue, the authors explore the applicability of concepts from the area of orthogonally persistent operating systems to computer forensics. In these systems, the state of the kernel and user space applications is captured and recorded periodically. For this reason, the solution

proposed by the authors requires integration in the operating system, thus resulting in no impact to practical systems. The first systematization of the atomicity concept was published by Vomel and Freiling [60] through a formal definition of three criteria, namely *correctness*, *integrity*, and *atomicity*. The authors pointed out that an atomic snapshot is a snapshot that "*does not show any signs of concurrent system activity*." Four years later, Gruhn and Freiling [24] returned to the subject, this time evaluating how those three criteria are respected by 12 different acquisition tools. More recently, Case and Richard [12] brought back under the spotlight the *page smearing* problem. The authors argue that—with the current acquisition tools—the smearing effect will become more and more common, since, nowadays, servers are equipped with a large amount of RAM that causes longer acquisition time. They also underline how page smearing "*is one of the most pressing issues*" in the field of memory forensics.

The very first software acquisition method that tried to obtain an atomic snapshot is BodySnatcher [51]. The underlining idea of this work is to freeze the running operating system by injecting a small custom acquisition OS. The approach, as acknowledged by the authors, has several severe limitations—including the fact of being very operating-system dependent and, in its current implementation, of supporting only systems with a single CPU core. In 2010, Forenscope [14] took advantage of the data remanence effect in memory chips to reboot the system and divert the boot sequence to the acquisition module. This approach is generally referred to as the cold boot memory acquisition [27]. Unfortunately, a few years later, an in-depth analysis of cold boot practicability in memory forensics [8] concluded that the data remanence strongly depends both on the chipset and on the memory modules used and that some combinations of components do *not* retain the content of the RAM upon a reboot.

The next attempt toward a sound memory acquisition was done by HyperSleuth [41], a tool based on the Intel virtualization technology. Working at this high-privileged level allows HyperSleuth to dump pages using two different strategies: *dump-on-write* and *dump-on-idle*. The first is triggered whenever a page is modified by the guest operating system, while the second is whenever the guest itself is in an idle state. While this is resilient against malware, which tries to evade the forensics acquisition, it does not avoid data structure inconsistencies. In 2012, Reina et al. presented `SMMDumper` [45], a special firmware running in System Management Mode (SMM). Entering this special mode ensures that the acquisition process is completely atomic, since the operating system has no chance to gain back the control. Even if this tool satisfies all of the three forensics criteria, it suffers from the fact that it cannot be hot-plugged in a running system, and therefore it needs to be pre-installed beforehand. A similar approach has also been proposed by Sun et al. [57] for the ARM architecture. In this case, the authors propose an acquisition process built on top of the `TrustZone` technology.

While not directly related to our findings, Bhatia et al. [6] and Saltaformaggio et al. [48–50] studied how it is possible to create a timeline of past user activities and to extract a number of photographic evidences and past Android app GUIs.

While all the attempts described above are interesting from a research perspective, many require the system to be carefully configured beforehand and none is mature enough to be used in real investigations or production systems. As a consequence, memory acquisition of non-virtualized systems continues to be performed using kernel-level solutions that copy the memory content while the system is running.

8 CONCLUSION

Kernel-level acquisition tools are the de facto standard for memory acquisition in traditional desktop and non-virtualized server systems. However, in this article, we show that the acquired data contains many small inconsistencies that can make the result of the following analysis unreliable.

For this reason, we argue that any analysis should take both the *spatial* and *temporal* aspects of the memory content into account. The first allows existing tools to locate relevant information and reconstruct high-level abstraction used by the operating system. The second, proposed for the first time in this article, allows the analyst to detect the presence of errors and estimate the reliability of the obtained results. Finally, we show that better acquisition tools can be developed that achieve local consistency in many important data structures and in the memory of userland processes.

REFERENCES

[1] Frank Adelstein. 2006. Live forensics: Diagnosing your system without killing it first. *Commun. ACM* 49, 2 (2006), 63–66.

[2] Noora Al Mutawa, Ibtesam Al Awadhi, Ibrahim Baggili, and Andrew Marrington. 2011. Forensic artifacts of Facebook's instant messaging service. In *2011 International Conference for Internet Technology and Secured Transactions (ICITST)*. IEEE, 771–776.

[3] Ali Reza Arasteh and Mourad Debbabi. 2007. Forensic memory analysis: From stack and code to execution history. *Digital Invest.* 4 (2007), 114–125.

[4] Michael Becher and Maximillian Dornseif. 2004. Feuriges hacken-spaß mit firewire. In *21C3: Proceedings of the 21st Chaos Communication Congress*, Vol. 10.

[5] Michael Becher, Maximillian Dornseif, and Christian N. Klein. 2005. FireWire: All your memory are belong to us. *Proceedings of CanSecWest* .

[6] Rohit Bhatia, Brendan Saltaformaggio, Seung Jei Yang, Aisha Ali-Gombe, Xiangyu Zhang, Dongyan Xu, and Golden G. Richard III. 2018. "Tipped off by your memory allocator": Device-wide user activity sequencing from android memory images. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS'18), San Diego.*

[7] Frank Block and Andreas Dewald. 2017. Linux memory forensics: Dissecting the user space process heap. *Digital Invest.* 22 (2017), S66–S75.

[8] Richard Carbone, C. Bean, and M. Salois. 2011. *An In-depth Analysis of the Cold Boot Attack: Can it be Used for Sound Forensic Memory Acquisition?* Technical Report. Defence Research and Development Canada Valcartier, Quebec.

[9] Brian D. Carrier and Joe Grand. 2004. A hardware-based memory acquisition procedure for digital investigations. *Digital Invest.* 1, 1 (2004), 50–60.

[10] Harlan Carvey. 2005. Digital forensics of the physical memory. Retrieved from http://seclists.org/incidents/2005/Jun/22.

[11] Andrew Case, Lodovico Marziale, Cris Neckar, and Golden G. Richard III. 2010. Treasure and tragedy in kmem_cache mining for live forensics investigation. *Digital Invest.* 7 (2010), S41–S47.

[12] Andrew Case and Golden G. Richard. 2017. Memory forensics: The path forward. *Digital Invest.* 20 (2017), 23–33.

[13] Andrew Case and Golden G. Richard III. 2016. Detecting objective-C malware through memory forensics. *Digital Invest.* 18 (2016), S3–S10.

[14] Ellick Chan, Shivaram Venkataraman, Francis David, Amey Chaugule, and Roy Campbell. 2010. Forenscope: A framework for live forensics. In *Proceedings of the 26th Annual Computer Security Applications Conference.* ACM, 307–316.

[15] M. Cohen. 2012. WinPMEM.

[16] Michael Cohen. 2014. Rekall memory forensics framework. *DFIR Prague.*

[17] Guilherme Cox, Zi Yan, Abhishek Bhattacharjee, and Vinod Ganapathy. 2018. Secure, consistent, and high-performance memory snapshotting. In *Proceedings of the 8th ACM Conference on Data and Application Security and Privacy Conference.* ACM, 236–247.

[18] Brendan Dolan-Gavitt. 2007. The VAD tree: A process-eye view of physical memory. *Digital Invest.* 4 (2007), 62–64.

[19] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. 2011. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *2011 IEEE Symposium on Security and Privacy (SP).* IEEE, 297–312.

[20] Brendan Dolan-Gavitt, Abhinav Srivastava, Patrick Traynor, and Jonathon Giffin. 2009. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM Conference on Computer and Communications Security.* ACM, 566–577.

[21] Qian Feng, Aravind Prakash, Heng Yin, and Zhiqiang Lin. 2014. Mace: High-coverage and robust memory analysis for commodity operating systems. In *Proceedings of the 30th Annual Computer Security Applications Conference.* ACM, 196–205.

[22] Mel Gorman. [n. d.]. Understanding the Linux Virtual Memory Manager. Retrieved from http://www.makelinux.net/books/lvmm/understand007#toc31.

[23] Mariano Graziano, Andrea Lanzi, and Davide Balzarotti. 2013. Hypervisor memory forensics. In *International Workshop on Recent Advances in Intrusion Detection.* Springer, 21–40.

[24] Michael Gruhn and Felix C. Freiling. 2016. Evaluating atomicity, and integrity of correct memory acquisition methods. *Digital Invest.* 16 (2016), S1–S10.

[25] Yufei Gu and Zhiqiang Lin. 2016. Derandomizing kernel address space layout for memory introspection and forensics. In *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy*. ACM, 62–72.

[26] Adrien Guinet. 2017. wannakey. Retrieved from https://github.com/aguinet/wannakey.

[27] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. 2009. Lest we remember: Cold-boot attacks on encryption keys. *Commun. ACM* 52, 5 (2009), 91–98.

[28] Brian Hay, Matt Bishop, and Kara Nance. 2009. Live analysis: Progress and challenges. *IEEE Secur. Privacy* 2 (2009), 30–37.

[29] Owen S. Hofmann, Alan M. Dunn, Sangman Kim, Indrajit Roy, and Emmett Witchel. 2011. Ensuring operating system kernel integrity with OSck. In *ACM SIGARCH Computer Architecture News*, Vol. 39. ACM, 279–290.

[30] Ewa Huebner, Derek Bem, Frans Henskens, and Mark Wallis. 2007. Persistent systems techniques in forensic acquisition of memory. *Digital Invest.* 4, 3–4 (2007), 129–137.

[31] Ryan Jones. 2007. Safer live forensic acquisition. *Computer Science Laboratory, University of Kent.*

[32] Jesse D. Kornblum. 2007. Using every part of the buffalo in Windows memory analysis. *Digital Invest.* 4, 1 (2007), 24–29.

[33] Stefan Le Berre. 2018. From corrupted memory dump to rootkit detection. Retrieved from https://exatrack.com/public/Memdump_NDH_2018.pdf.

[34] Marthie Lessing and Basie Von Solms. 2008. Live forensic acquisition as alternative to traditional forensic processes. In *International Conference on IT Incident Management & IT Forensic.*

[35] Eugene Libster and Jesse D. Kornblum. 2008. A proposal for an integrated memory acquisition mechanism. *ACM SIGOPS Operating Systems Review* 42, 3 (2008), 14–20.

[36] Michael Hale Ligh, Andrew Case, Jamie Levy, and Aaron Walters. 2014. *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory.* John Wiley & Sons.

[37] Zhiqiang Lin, Junghwan Rhee, Xiangyu Zhang, Dongyan Xu, and Xuxian Jiang. 2011. SigGraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *NDSS.*

[38] Holger Macht. 2013. Live memory forensics on android with volatility. *Friedrich-Alexander University Erlangen-Nuremberg.*

[39] Mandiant. [n. d.]. Memoryze.

[40] Jean Marsault. 2017. Volatility-notpetyakeys. Retrieved from https://github.com/Iansus/Volatility-notpetyakeys.

[41] Lorenzo Martignoni, Aristide Fattori, Roberto Paleari, and Lorenzo Cavallaro. 2010. Live and trustworthy forensic analysis of commodity production systems. In *RAID.* Springer, 297–316.

[42] Robert J. McDown, Cihan Varol, Leonardo Carvajal, and Lei Chen. 2016. In-depth analysis of computer memory acquisition software for forensic purposes. *J. Forensic Sci.* 61 (2016), S110–S116.

[43] Andreas Moser and Michael I. Cohen. 2013. Hunting in the enterprise: Forensic triage and incident response. *Digital Invest.* 10, 2 (2013), 89–98.

[44] Yuto Otsuki, Yuhei Kawakoya, Makoto Iwamura, Jun Miyoshi, and Kazuhiko Ohkubo. 2018. Building stack traces from memory dump of Windows x64. *Digital Invest.* 24 (2018), S101–S110.

[45] Alessandro Reina, Aristide Fattori, Fabio Pagani, Lorenzo Cavallaro, and Danilo Bruschi. 2012. When hardware meets software: A bulletproof solution to forensic memory acquisition. In *Proceedings of the 28th Annual Computer Security Applications Conference.* ACM, 79–88.

[46] Nicolas Ruff. 2008. Windows memory forensics. *J. Comput. Virol.* 4, 2 (2008), 83–100.

[47] Brendan Saltaformaggio. 2018. Convicted by Memory: Recovering spatial-temporal digital evidence from memory images. USENIX Association, Atlanta, GA.

[48] Brendan Saltaformaggio, Rohit Bhatia, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu. 2015. GUITAR: Piecing together android app GUIs from memory images. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security.* ACM, 120–132.

[49] Brendan Saltaformaggio, Rohit Bhatia, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu. 2015. Vcr: App-agnostic recovery of photographic evidence from android device memory images. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security.* ACM, 146–157.

[50] Brendan Saltaformaggio, Rohit Bhatia, Xiangyu Zhang, Dongyan Xu, and Golden G. Richard III. 2016. Screen after previous screens: Spatial-temporal recreation of android app displays from memory images. In *USENIX Security Symposium.* 1137–1151.

[51] Bradley Schatz. 2007. BodySnatcher: Towards reliable volatile memory acquisition by software. *Digital Invest.* 4 (2007), 126–134.

[52] Andreas Schuster. 2006. Searching for processes and threads in Microsoft Windows memory dumps. *Digital Invest.* 3 (2006), 10–16.

[53] Matthew Simon and Jill Slay. 2010. Recovery of skype application activity data from physical memory. In *2010 5th International Conference on Availability, Reliability, and Security.* IEEE, 283–288.

[54] Matthew Phillip Simon and Jill Slay. 2011. Recovery of pidgin chat communication artefacts from physical memory: A pilot test to determine feasibility. In *2011 6th International Conference on Availability, Reliability and Security*. IEEE, 183–188.
[55] Arkadiusz Socała and Michael Cohen. 2016. Automatic profile generation for live Linux Memory analysis. *Digital Invest.* 16 (2016), S11–S24.
[56] Johannes Stüttgen and Michael Cohen. 2013. Anti-forensic resilient memory acquisition. *Digital Invest.* 10 (2013), S105–S115.
[57] He Sun, Kun Sun, Yuewu Wang, and Jiwu Jing. 2015. Reliable and trustworthy memory acquisition on smartphones. *IEEE Trans. Inf. Forensics Secur.* 10, 12 (2015), 2547–2561.
[58] Joe Sylve. 2012. Lime-linux memory extractor. In *Proceedings of the 7th ShmooCon Conference*.
[59] Stefan Vömel and Felix C. Freiling. 2011. A survey of main memory acquisition and analysis techniques for the windows operating system. *Digital Invest.* 8, 1 (2011), 3–22.
[60] Stefan Vömel and Felix C. Freiling. 2012. Correctness, atomicity, and integrity: Defining criteria for forensically-sound memory acquisition. *Digital Invest.* 9, 2 (2012), 125–137.
[61] Stefan Vömel and Johannes Stüttgen. 2013. An evaluation platform for forensic memory acquisition software. *Digital Invest.* 10 (2013), S30–S40.
[62] Aaron Walters. 2007. The volatility framework: Volatile memory artifact extraction utility framework.

Received August 2018; accepted December 2018