An Experimental Assessment of Inconsistencies in Memory Forensics

JENNY OTTMANN, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany
FRANK BREITINGER, University of Lausanne, Switzerland
FELIX FREILING, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Memory forensics is concerned with the acquisition and analysis of copies of volatile memory (memory dumps). Based on an empirical assessment of observable inconsistencies in 360 memory dumps of a running Linux system, we confirm a state of overwhelming inconsistency in memory forensics: almost a third of these dumps had an empty process list and was therefore obviously incomplete. Out of those dumps that were analyzable, almost every second dump showed some form of inconsistency that potentially impacts the interpretation of the dump in a forensic investigation. These results are based on a new way to estimate the level of causal consistency of a memory dump. The factors influencing these inconsistencies are less clear but in general correlate with the level of concurrency (system load and number of threads).

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

Additional Key Words and Phrases: Memory forensics, memory acquisition, inconsistencies, memory analysis

ACM Reference format:
Jenny Ottmann, Frank Breitinger, and Felix Freiling. 2023. An Experimental Assessment of Inconsistencies in Memory Forensics. *ACM Trans. Priv. Sec.* 27, 1, Article 2 (December 2023), 29 pages.
https://doi.org/10.1145/3628600

1 INTRODUCTION

Digital evidence—that is, data stored on digital media with relevance to a criminal offense [5]—is of vital importance in investigations and incident management. Sources of digital evidence are still predominantly persistent storage devices which become increasingly harder to access due to the use of trusted hardware and encryption. In addition, incident response must deal with advanced malware that resides in memory only and leaves traces on disk in encrypted form, if traces are left at all.

From the viewpoint of forensic computing, these challenges can be mitigated doing *memory forensics*. This means that not only persistent storage is acquired but also a copy of physical main memory is taken as a so-called *memory dump*, resulting in an increasing necessity of reliable tools and methods to acquire and analyze such memory dumps.

This work was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) as part of the Research and Training Group 2475 “Cybercrime and Forensic Computin” under grant 393541319/GRK2475/1-2019.
Authors' addresses: J. Ottmann and F. Freiling, Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Erlangen, Germany; e-mails: {jenny.ottmann, felix.freiling}@fau.de; F. Breitinger, School of Criminal Justice, University of Lausanne, Lausanne, Switzerland; e-mail: frank.breitinger@unil.ch.

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in ACM Transactions on Privacy and Security, Vol. 27, No. 1, Article 2.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
https://doi.org/10.1145/3628600

There are different ways to acquire main memory, but advances in technology have narrowed down the spectrum of methods that were available 10 years ago [36] to two main approaches which we characterize as *clean* and *dirty*. In the *clean* approach, the system can be halted, and memory can be accessed by independent means. This is possible if the system runs as a **Virtual Machine (VM)** [20] or if the memory hardware is modified to offer such specific functionality [9]. When halting the system, all system activities on main memory are frozen and the contents of main memory can conveniently be written to a file. Another approach to clean memory acquisition is the application of cold boot techniques [23].

When cold boot techniques cannot be used, the clean approach requires privileged access to the VM monitor or non-standard hardware. If this is not available, the established method is to execute post-hoc kernel level dumping, an approach that is inherently self-referential because the dumping software accesses its own memory. This approach is characterized as *dirty* because the system and user processes remain active during the acquisition of the memory dump. It is well known that the concurrency between these activities leads to inconsistencies such as page smearing [4] because the contents of the acquired pages continue to change.

To illustrate the problem, consider a system where low physical addresses store the contents of user programs and high addresses contain kernel code and data. Now assume that the code and data of user programs might have already been acquired at the beginning of the acquisition. If a malware infection occurs, new threads are created in data structures that are subsequently also written to the memory dump. However, the executable code of the malware, which is stored in low addresses, will be missing in the memory dump, potentially misleading criminal investigators to the wrong conclusion that no malware was present. Therefore, even a correctly working memory dumping software presents an investigator with an inconsistent view of physical memory, causing problems during structured analysis or completely hampering it. An in-depth understanding of how consistent a memory dump is and how inconsistencies may affect the interpretation of this form of digital evidence is vital for criminal investigations and incident response.

For this work, we constructed an automated evaluation method and decided on a comparatively small memory size of 4 GB for our experiments and a minimal system configuration. Setting the size to 4 GB ensures that results are obtained in a feasible amount of time and, more importantly, allows us to share the dataset. Additionally, to enable precise control of the degree of concurrently running processes and their memory usage, benchmarking processes were chosen to vary the system load. The accumulated knowledge can be used when setting up experiments with other tools or for evaluation environments that mimic real-world scenarios.

1.1 Related Work

Previous work has attempted to determine the quality of memory acquisition tools. For example, Inoue et al. [18], Campbell [3], and Lempereur et al. [24] compared the memory dumps produced by the tools with the memory contents of the system on which they were executed. Their findings show that some tools impact memory contents more than others. Additionally, they highlight the importance and difficulty of obtaining ground truth.

Later, Vömel and Freiling [37] defined formal quality criteria, and the one this work builds on is called *atomicity*. It describes inconsistencies on the causal level. Intuitively, a memory dump is atomic if the cause-effect relationships between the memory regions are consistent. If a memory dump is not atomic, inconsistent states might be present in the memory dump (e.g., between PT entries and the actual contents of the physical pages).

A practical evaluation of memory acquisition tools using the criteria defined by Vömel and Freiling [37] was performed by Vömel and Stüttgen [38] shortly after the conceptualization of the criteria. They used Bochs as the testing environment and took a white-box testing approach,

relying on inserting hypercalls into the tools' code. Consequently, their evaluation was limited to tools for which source code was available. In their evaluation, the degree of atomicity could only be estimated. They observed a higher percentage of pages for which atomicity was possibly violated for larger memory sizes. Gruhn and Freiling [15] took a black-box approach to test memory acquisition tools to evaluate techniques without having access to their source code. However, the level of atomicity a method achieves could only be approximated. Overall, the results confirmed that methods that allow freezing the state of the system achieve the highest atomicity.

Pagani et al. [31] introduced several new inconsistency definitions which were motivated by scenarios that could occur when memory contents change while they are being copied. In all of them, a mismatch between the contents obtained for the respective memory regions and the contents that were coexistent in memory at some point in time occurs. As an example of such mismatches that could potentially hamper the analysis of processes contained in the memory dump, they took a closer look at the data structures of the Linux kernel and compared the number of stored **Virtual Memory Areas (VMAs)** of a process in two different kernel data structures to the VMA counter of the process. They report that inconsistencies were found in more than 80% of the analyzed snapshots and that in some cases the code and/or stack segment of processes could not be found with Volatility [26].

In subsequent work, Ottmann et al. [29] added two more consistency definitions that narrow the gap between the notions of causal consistency and the notion of a completely frozen system: *instantaneous consistency* and *quasi-instantaneous consistency*. Instantaneous consistency describes the ideal case for memory acquisition, the system can be frozen, and no memory contents change during the acquisition. Quasi-instantaneous consistency is a concept that respects causal order but is indistinguishable from a frozen system. When a memory dump is quasi-instantaneously consistent, a point in time during the acquisition can be found at which the memory contents were in the same state as the acquired ones. A method to observe quasi-instantaneous consistency was presented recently [30].

Distinction to Other Areas of Research. The preceding consistency definitions are not to be confused with those of shared memory models. Such models define the views that concurrent processes can have on shared memory. The classical notion of *linearizability* (also known as *atomic consistency*) [16], for example, demands that all processes see the same "real time" order of write operations in the system, whereas *causal consistency* [17] allows processes to see different orders as long as they respect *potential causality* [22]. Unfortunately, inconsistent snapshots of memory can still occur even if the memory model satisfies causal consistency, as the preceding example shows.

Our work is related to the large body of results that investigate the creation of snapshots in distributed concurrent systems often with the aim to detect predicates on the state of a distributed computation [6, 7, 8, 12, 32]. In this line of research, it is usually assumed that all read and write accesses to memory can be intercepted (or at least monitored). We do not make this assumption. This also holds for work that relates the consistency criteria of shared memory to the possibility of creating *atomic snapshots* [1].

1.2 Contributions

Although there are numerous concepts to define different forms of consistency of memory dumps, practitioners regularly report that many memory dumps are in some unknown form inconsistent, and even sophisticated analysis tools like Volatility fail, for example, by producing no or a misleading output. Nevertheless, we have (1) no empirical evidence confirming this feeling of *overwhelming inconsistency*, and (2) we know surprisingly little about in what way practical memory dumps are inconsistent and the factors that influence the consistency.

This work attempts to quantitatively assess the general types of inconsistencies that occur using state-of-the-art memory acquisition and analysis. Since inconsistencies are fueled by concurrency, we also empirically explore the influence of "concurrency enabling" factors on the level of inconsistency such as the number of CPUs, number of concurrent threads, and system load. Overall, this work provides the following contributions:

(1) We propose a new way to estimate the level of *causal consistency* of a memory dump by injecting a *pivot process* that generates predictable patterns from which causal relations can be inferred using concepts of virtual time from concurrency theory.
(2) We establish a hierarchical classification of memory dumps from the viewpoint of practical *analyzability*. This classification is based on the ability to perform certain steps of *structured analysis* in memory forensics and therefore assesses observable effects of inconsistency (without explaining its cause). The classification ranges from memory dumps that are obviously *incomplete* to ones that allow address space extraction and analysis of the pivot process to assess *causal consistency*.
(3) Using this classification, we perform an empirical assessment of the practical consequences of observable inconsistencies based on 360 memory dumps of a running Linux system, thereby confirming a state of *overwhelming inconsistency* in memory forensics:
    - Out of these 360 dumps, 102 were meaningless in the sense that they were incomplete (the process list was empty).
    - Out of the 258 valid (i.e., analyzable) memory dumps, for each combination of parameter settings we randomly selected 20 dumps, resulting in a dataset of 180 dumps. Overall, 80 out of 180 contained VMA inconsistencies and 73 out of 180 showed causal inconsistencies within the pivot program.
(4) The factors influencing these inconsistencies are less clear, but in general, the number of inconsistencies correlates with the level of concurrency:
    - The main influencing factor regarding VMA inconsistencies appears to be system load. Although no system load resulted in no VMA inconsistencies, with a high system load, 54 out of 60 valid memory dumps contained VMA inconsistencies.
    - Causal inconsistencies appear to correlate not with load but with the number of threads. If one thread is used, 8 out of 60 memory dumps showed causal inconsistencies. If two or four threads were used, 35 and 30 dumps were causally inconsistent.
(5) We make the 180 analyzed memory dumps and the data gathered about meaningless dumps publicly available [28].

All experiments were performed using current hardware and state-of-the-art memory acquisition and analysis tools (LiME [21] for memory acquisition and Volatility [26] for memory analysis; for more details on the tools, see Section 2).

1.3 Outline

The remainder of the article is structured as follows. First, we provide background information on memory forensics and causal consistency in Section 2. Then, the experimental design is described in Section 3. This is followed by the experimental results in Section 4 and their discussion in Section 5. Finally, Section 6 summarizes our conclusions and gives an outlook on future work.

2 MEMORY FORENSICS AND CAUSAL CONSISTENCY

This section first provides the background on how memory acquisition at the kernel level works (e.g., LiME dumps). Then, structured memory analysis is explained, and an overview of the Volatility functionalities we use for our evaluation is given. Last, the definition of causal consistency is

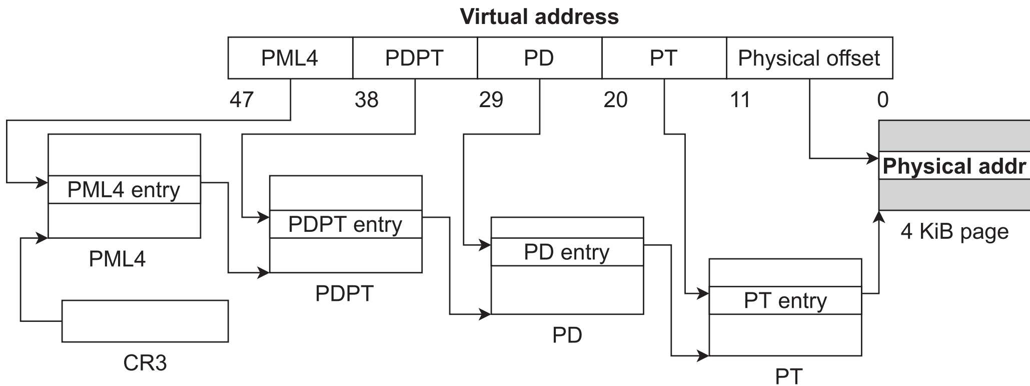


Fig. 1. Translation of a virtual to a physical address with page size of 4 KiB. During address translation, the address of a process's PML4 is read from control register CR3. Then, the corresponding part of the virtual address is used to identify the entry which points to the address of the PDPT. In each structure in the hierarchy, this is repeated with a different part of the virtual address, until the entry in the PT identifies the address of the page frame. Finally, the lowest 12 bits of the virtual address are added, resulting in the physical address [19, ch. 4, p. 20].

presented, and a possible method to observe the causal consistency in a set of memory regions is explained.

2.1 Kernel Level Memory Dumping

Memory acquisition tools operating at the kernel or OS level dump the memory contents concurrently with the system activity, and therefore we characterize them as dirty (see Section 1). They proceed sequentially from one end of the physical address space to the other. Before the sequential acquisition is started, the tool needs to determine which parts of the physical address space are available to the OS as RAM and which are reserved. Reserved ranges are used for device DMA and accessing them can lead to system crashes. Under Linux, the ranges can be identified by examining the symbol `iomem_resource` [33].

Virtual address spaces allow to isolate different processes from each other while allowing them to allocate more memory than is available in the physical address space. The virtual memory is split into pieces of the same size called pages. A process can see and access only the pages of its own continuous virtual address space. Although the virtual addresses of the pages are continuous, their counterparts in the physical address space, page frames, do not have to be located consecutively. Paging data structures are used to store mappings from virtual to physical addresses. When a virtual address is accessed, the memory management unit uses them to identify the corresponding page frame [34, p. 194–198]. On x86-64 [19, ch. 2, p. 6], four levels of paging data structures are used to translate a virtual to a physical address when the page size is 4 KiB. The entry point is the **Page Map Level 4 (PML4)**. It contains references to **Page Directory Pointer Tables (PDPTs)**, which in turn contain references to **Page Directories (PDs)**. These contain references to the last data structure the **Page Tables (PTs)**. Each entry in a PT references a page frame. In addition to the addresses of the next paging data structure or page frame, the entries in all translation levels also contain access rights and other information for memory management. The physical address of a process's PML4 is loaded into the `CR3` control register when execution switches to it. Then the address translation from virtual to physical is performed by the memory management unit. Figure 1 summarizes the translation process from a virtual to a physical address. Because they play such an important role in reconstructing the virtual address spaces, the paging data structures are a vital part of the memory dump for analysis.

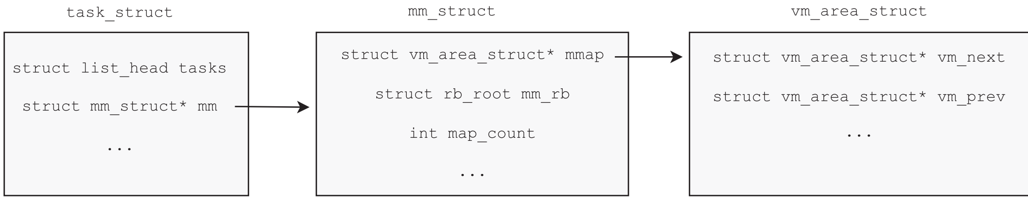


Fig. 2. Structures used in the Linux kernel to organize processes and their memory. For each process, a `task_structure` is created. Information about the process's memory can be found in its `mm_struct`. The individual VMAs are organized in a list of structures of type `vm_area_struct`. All structures are shown according to the kernel version used in the experiment.

A central data structure in the Linux kernel is the *process list* consisting of a sequence of *process control blocks*, one for each active process in the system. In Linux, a process control block corresponds to a structure of type `task_struct` [2, p. 81] (Figure 2). These task structures form a linked list using the `list_head` structures embedded in each of them. The list starts with the `init_task`, the first task that is created during the initialization of the kernel [2, p. 89].

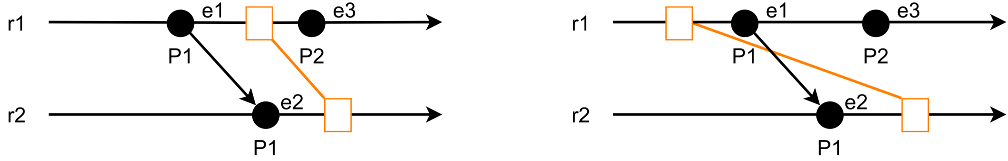
The different memory regions a process allocates are called *virtual memory areas* (VMAs). These are maintained by the Linux kernel as a linked list attached to the process control block and organized as follows. The `mm` member of the task structure points to a structure of type `mm_struct` which gathers all information about the memory assigned to the process. VMAs are represented by structures of type `vm_area_struct`. The VMA structures assigned to the task can be found by following the linked list starting with the `mmap` pointer in the `mm_struct` which points to the first element in the list. Additionally, the VMA structures are saved in a red-black tree in the member `mm_rb`. The total number of VMAs is also saved in the `mm_struct` in the integer `map_count` [2, pp. 353–362].

The idea of kernel level memory dumping is to map physical addresses systematically and sequentially into the kernel address space by creating PT entries that resolve a kernel virtual address to the desired physical address. This can be done with the help of kernel APIs (e.g., `kmap` in Linux). Alternatively, the paging hierarchy traversal, as shown in Figure 1, and manipulation of the PT entry can be implemented from scratch to reduce the reliance on the potentially untrusted OS [33]. LiME relies on kernel APIs to create the mappings.

2.2 Structured Analysis in Memory Forensics

The simplest approach to analyze a memory dump is to search the dump for occurrences of specific strings. Although this can lead to helpful results if the keywords are well chosen, the *context* of the found strings (e.g., the process address space it belongs to is generally unknown. Therefore, this method is called *unstructured*.

Structured analysis offers a more sophisticated way to gather relevant data from a memory dump. It relies on OS specific management data structures to reconstruct the state of the system at the time of the memory acquisition. This allows us to reproduce the context of found information. For example, if a process of interest is found, it is possible to find its parent process or reconstruct its address space. Detailed knowledge about the inner workings of a specific OS version and the used data structures is necessary to enable this. This information is often not well documented and changes frequently. Because of changes depending on specific OS versions or configurations, tools like Volatility rely on *profiles* to provide the necessary information for the analysis. Profiles are created for specific OS versions and contain information allowing us to find and parse OS data structures. In the case of Linux, a profile contains the specific kernel's debug symbols and data structures.



(a) The snapshot is consistent because not only e_2 but also its cause e_1 is included

(b) The snapshot is inconsistent because the cause of e_2, e_1 is missing

Fig. 3. In the space time diagram, two memory regions r_1 and r_2 can be seen. Their change over time is shown on the arrows with time passing from left to right. Accesses to the regions are called *events* and denoted as black dots labeled with *e*. Should the events be executed by more than one process, the process number x is denoted together with the event as P_x.

Currently, Volatility [26] is one of the most advanced tools for structured memory analysis. It organizes its functionality as modules (called *plugins*). This work makes use of the following standard plugins:

With the `linux_pslist` *plugin*, the process list is reconstructed following the pointers to the next `task_struct` starting with the `init_task` (see Section 2.1). Even when a process has finished executing and is removed from the process list, its `task_struct` might still exist in the memory dump.

With the `linux_psscan` *plugin*, such unlinked structures can be found. Using the knowledge about the structure's layout contained in the profile, data patterns are searched in the memory dump that could be `task_struct` structures. Then, the plugin checks if the found struct's pid is in the expected range and if the process state is a valid one to confirm that the found pattern can be correctly interpreted as a `task_struct`.

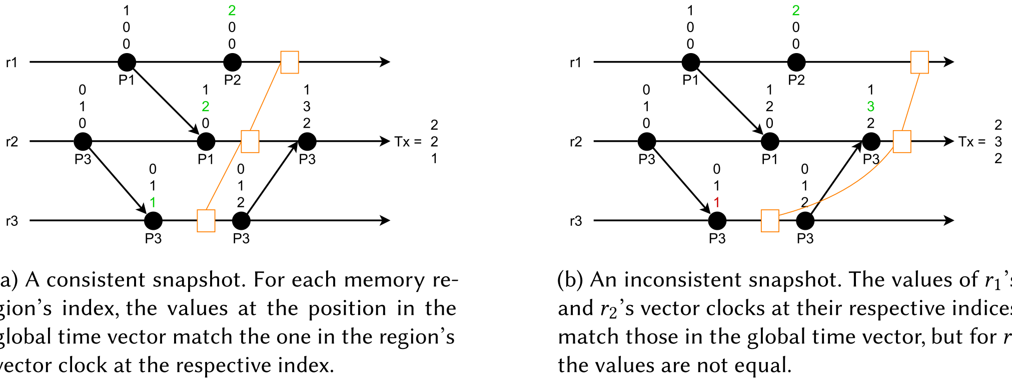
With the `linux_psxview` *plugin*, a comparative view of processes recovered by different plugins, among others, `linux_pslist` and `linux_psscan`, can be obtained.

With the `linux_dump_map` *plugin*, the memory allocated to processes contained in the process list can be extracted. This requires the translation of the virtual addresses of VMAs that were allocated by the processes into physical ones. This is done by interpreting the paging structures and accessing the corresponding parts of the memory dump.

2.3 Defining and Observing Causal Consistency

The notion of *causal consistency* (originally called *atomicity* by Vömel and Freiling [37] and renamed by Ottmann et al. [29] for a clearer contextual differentiation from the other consistency models) evaluates if the state of causally dependent changes on memory regions is consistent. Here, consistency is assessed with regard to causally dependent changes to memory regions. The causal dependencies are constructed by the order in which processes access memory regions. The remainder of the section summarizes the more detailed explanation of causal consistency and its observability given in the work of Ottmann et al. [29].

To visualize causal relationships, space-time diagrams are used as depicted in Figure 3. The horizontal arrows r_1 and r_2 represent memory regions over time. The black dots represent events indicating write accesses to the memory regions. In the example, three events, e_1, e_2, and e_3, are shown; e_1 and e_2 are executed by process P_1 and e_3 by process P_2. In the example, P_1 first executes e_1 and then e_2. Therefore, an arrow starting at e_1 and ending at e_2 connects the two events. Consequently, the arrow represents a *possible* causal relationship between e_1 and e_2 [22]. For example, with event e_1, a pointer to an address in region r_2 could be set and with e_2 the contents at this address could be changed. Or in e_1, data could be written to the first half of the buffer and in e_2 to



(a) A consistent snapshot. For each memory region's index, the values at the position in the global time vector match the one in the region's vector clock at the respective index.

(b) An inconsistent snapshot. The values of r_1's and r_2's vector clocks at their respective indices match those in the global time vector, but for r_3 the values are not equal.

Fig. 4. Vector clocks keep track of the local order of events as well as connections to other memory regions. The vector's size equals the number of observed memory regions. Each region has a unique index in the vector that serves as its local counter that is incremented when an event happens. Processes carry the values of the vector of their last accessed region with them, which are used when the vector of the next visited region is updated.

the second half. The process of copying memory contents is shown in orange where a rectangle shows the point in time at which the region was dumped. Figure 3(a) shows a causally consistent memory acquisition. r_1 is copied after e_1 has been executed, then e_2 happens and r_2 is copied; last, e_3 is executed. This order respects the cause-effect relationship between e_1 and e_2. In contrast, the memory dump shown in Figure 3(b) is causally inconsistent because of a small change in the order of memory acquisition and execution of events. Here, r_1 is copied before e_1 happens instead of the other way around. Because e_1 happened before e_2 *and* is missing in the memory dump, this dump is causally inconsistent. If the two events were not executed by the same process but concurrent with each other, causal consistency would not be violated.

Causal relations between memory regions can be tracked using the concept of *vector clocks* [27]: each memory region is assigned a vector that contains one row for each surveyed memory region. All memory regions get a unique index into the vector assigned. The value of this index is the region's local counter of events. It is incremented by one each time an event is executed on the region. The other indices are updated whenever a process executes an event on a different region than the one it was accessed before. For this purpose, the processes save the vector clock of the region they accessed last. When they execute an event on a different region, the local counter is incremented by 1. The local and saved vector clocks are compared and the values of the local one are updated to the higher value at each index. Intuitively, the value of a vector clock represents the *causal past* of an event: it is a compact expression of which events have been observed and which ones have not. By comparing the vector clocks of the different regions, a partial order of events can be constructed and causal relationships identified.

Figure 4 shows an example execution with two different memory snapshots. The index of region r_1's local counter is 1, r_2's is 2, and r_3's is 3. To illustrate the update process, we follow the actions of P_3 from left to right. First, P_3 executes an event on r_2 setting its vector from initially all zeros to $(0, 1, 0)$. When it subsequently accesses r_3, r_3's local counter is updated to 1, and the vector now contains $(0, 1, 0)$. Then, the vector P_3 saved after the event on r_2 is compared to this vector and for every index the higher value is kept, resulting in the final values $(0, 1, 1)$. Next, P_3 accesses r_3 again, incrementing only the local counter. When it moves on to r_2, the local counter is set to 3. The local vector $(1, 3, 0)$ and the saved vector $(0, 1, 2)$ are combined, resulting in the new values $(1, 3, 2)$ for r_2's vector clock.

To identify an inconsistency, the *global time* vector t_s of the snapshot s must be calculated. This vector consists of the highest values for each index in all vector clocks $C_1, \ldots, C_n$ in the memory dump [27] as

$$t_s = sup(C_1, \ldots, C_n).$$

The function *sup* returns the supremum of the vectors—that is, the maximum for each index over all vectors $C_1, \ldots, C_n$ [27].

Next, each region's vector clock is compared to the global time. Only the values of the regions' vector clocks at their respective indices are compared to t_s at the same index. Snapshot s is consistent iff $t_s = (C_1[1], \ldots, C_n[n])$ [27].

Figure 4(a) shows an example of a consistent snapshot. Comparing the global time to the regions' vector clocks shows that for all memory regions, the value at the respective index is equal to the global time vector at the same index. Thus, for all regions, the causing event of the latest access on them is included in the snapshot. However, Figure 4(b) shows an example of an inconsistent snapshot. In this case, the last event on r_3 is missing from the snapshot. This is a problem as the last event on r_2, which is included in the snapshot, is causally dependent on the event. Therefore, the vector clock has not been updated yet and the inconsistency can be identified by comparing the vector clock to the global time vector.

Intuitively, a causally consistent memory dump can be acquired concurrently with normal system operation as long as the sequence in which memory regions are acquired does not violate the causality relation. To express that the system is "frozen" during the acquisition of memory, stricter consistency models must be used, such as *instantaneous consistency* [29].

3 EXPERIMENTAL SETUP

The objective of this study is to analyze the likeliness of the occurrence of inconsistencies under different degrees of concurrent activity. For this study, we focus on kernel level acquisition which is often applied but also frequently suffers from concurrent activity. The tool we used to create the memory dumps is LiME v1.9.1 [21], which is implemented as loadable kernel module. LiME was chosen as it operates at the kernel level, is open source, and any analysis results based on its usage can be freely shared. It is one of the few freely available memory acquisition tools for Linux. However, in the past years, it has only been maintained sporadically. An alternative is Microsoft AVML,[1] and contrary to LiME, it operates at the user level and uses either `/dev/crash`, `/proc/kcore` or `/dev/mem` as sources for the acquisition. Therefore, it is limited to systems on which the kernel feature `kernel_lockdown` is not enabled.

All experiments were executed in a VM with 4 GB of RAM running Ubuntu 18.04.2 LTS, Linux kernel 4.15.0-177-generic. A minimal system installation without GUI was chosen. No configurations regarding RAM usage were changed (e.g., swapping was *not* disabled). The number of CPUs was initially set to 2 but changed to 4 after the pre-study. Apart from the script used to control the execution of a pivot program and the loading of LiME, the system load was changed using the command line tool `stress` [35]. The purpose of the pivot program, the different load levels, and the parameters necessary for the experiments are explained in the subsequent sections.

The *pivot program* enables the investigation of causal inconsistencies in a part of the main memory. Setting up a system to track the causal relationships between physical pages should be possible but would require modifications at kernel and/or hypervisor level and possibly introduce a high system overhead. Instead, we concentrate on the causal relationships within one pivot program—more precisely, the causal relationships between data structures located on a process's heap.

[1] Acquire Volatile Memory for Linux: https://github.com/microsoft/avml

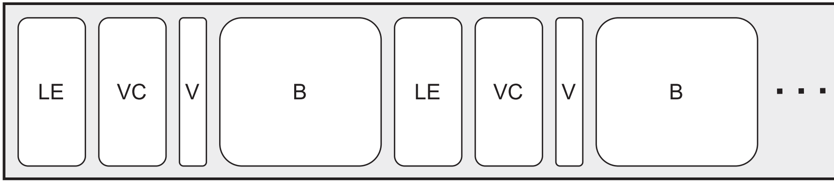


Fig. 5. In the pivot process's heap after a list element (LE), first the vector clock structure (VC), then the vector (V), and finally the buffer (B) follows. The sequence is the same for every allocated list element.

The program initializes a synchronized linked list (one thread). Next, several threads start to remove elements from the list at random positions, pause, and reinsert them behind the head element. Each list element has a vector clock that is updated when a thread removes or inserts a list element, thereby allowing to track the causal relationships between the list elements. Between the different list elements, a buffer is allocated to enforce a constant distance between the list elements. The buffer allows to enlarge the heap of the process. Its size was chosen to be 4,048 bytes, the page size in the test environment minus the size of the list element and vector clock structures. This ensures a distance of at least one page between subsequently allocated list elements. The heap layout, determined by the allocation order, is shown in Figure 5. Glibc's heap implementation is used. The number of list elements and the number of threads are passed as parameters when the program is started.

It is unlikely that the analysis of causal inconsistencies in the list will contribute to a better understanding of inconsistencies in other parts of the system. But it provides insight into how parallel activity in the process itself might influence the consistency of the acquired heap. Additionally, the analysis of the pivot process's heap serves as an example of the obstacles that might arise during the analysis of a simple (no relocations, no child processes) user process.

3.1 Parameters

Each test run uses two parameters: system load generated additionally to the execution of the pivot program and the number of threads that are active in the pivot program. To observe the influence of different degrees of concurrent system activity, the number of threads was set to 1, 2, and 4; system load was generated on top by executing the tool `stress` for 120 seconds with different parameters. Thereby, we increased the memory and CPU usage. We define three different levels of system load:

(1) *None*: No added activity.
(2) *Medium*: `stress --vm 2 -t 120`.
(3) *High*: `stress --vm 4 -c 2 -t 120`.

For both system load levels, the duration of the execution of `stress` is specified by passing the option `-t`. For medium system load, the `--vm` option is used to start two processes that repeatedly call `malloc` and `free`. The high system load level uses the same option but with four processes, and additionally two processes are started with the `-c` option to repeatedly call the `sqrt()` function. Whereas the first option focuses on the average memory usage, the latter option mostly imposes more load onto the CPUs. The two processes are added to increase the number of running processes without further increasing the memory load.

3.2 Method

To create the memory dumps, the same steps were executed on the VM. First, the pivot program is started with 100 list elements and varying threads as described earlier. Depending on the chosen

system load level, stress is started as well, and the memory contents are dumped using LiME. The execution time of LiME is recorded, and the main memory load is captured with the command line program free. The load is grabbed 15 times during the execution with an interval of 8 seconds using the command free -h -c 15 -s 8. In more detail, for all parameter combinations, the following procedure was executed automatically:

(1) Reboot the VM.
(2) Wait 60 seconds.
(3) Start the pivot program.
(4) Wait 60 seconds.
(5) If a load level other than *none*, start stress with appropriate parameters.
(6) Load LiME, and capture execution time using bash command time.
(7) Execute free periodically, and capture output,
(8) Wait for LiME to finish,
(9) Continue with step 1.

The VM is rebooted before each experiment execution to ensure an identical system state. To avoid errors, the manual interaction with the system is limited to the first reboot. For automation, a cronjob, executed automatically upon reboot, is used to start the experiment. When the desired number of iterations is reached, the bash script notifies a remote machine about the successful execution. The memory dumps are named according to the following scheme: *pid of pivot process - start address of heap - end address of heap - number of list elements - number of threads - execution number*. The start and end address of the process's heap are determined by parsing the contents of /proc/*pid of pivot process*/maps and extracting the address in the line where the pathname equals "[heap]."

3.3 Analysis

To identify inconsistencies, the following points are investigated:

- Inconsistencies regarding the number of VMAs allocated by a process
- Capability to extract the heap of the pivot program
- *If heap extraction fails*: Inconsistencies in the process list
- *On extracted heap*: Causal inconsistencies.

Whereas the first three points examine inconsistencies in kernel data structures, the last point concerns those in the heap of the process that are monitored using vector clocks. Inconsistencies in the kernel's data structures are identified using Volatility 2.6 [10].

The Volatility plugin described by Pagani et al. [31], linux_validate_vmas,[2] makes use of the different ways in which information about a process's VMAs is saved. It compares the number of VMAs in the list and the red-black tree with the total number of VMAs. If the numbers are different, an inconsistency has been found. For each memory dump, the total number of inconsistencies found with the plugin is noted. To get to each process's mm_struct, the linked list of processes needs to be recovered from the memory dump. Then for each process, the address of the mm_struct can be extracted from the task_struct.

Extracting the heap of our pivot process from the memory dumps also relies on the list of tasks and the task structure itself. The Volatility plugin linux_dump_map is used to extract the pages allocated for the process's heap. The appropriate address range is taken from the name of the memory dump (see Section 3.2). If the heap extraction fails, the state of the task list is

[2]Published by the authors under https://github.com/pagabuc/atomicity_tops

Table 1. Problems Observed in the Created Memory Dumps with the Different Combinations of Load Level and Thread Number

Threads	Load	No suitable address space	No heap	Volatility assertion error	<4 GB
	none	3	8	–	–
1	medium	3	13	–	4
	high	3	12	1	2
	none	9	18	–	10
2	medium	9	20	–	14
	high	9	18	1	11

investigated in more detail using the plugin `linux_psxview`. Both plugins' functionality is described in Section 2.2.

If the heap extraction succeeds, causal inconsistencies between the list elements are determined based on the vector clocks of the list elements. The list elements and vector clocks are extracted from the heap using a Python script that carves for the hex values at the beginning and end of each of the structures. List elements and vector clocks are matched using the address of the vector clock saved in the list element structure. After all vector clocks are gathered, the global time is computed as described in Section 2.3 and causal inconsistencies can be identified. The number of inconsistencies between the vectors and the global time is saved for each dump. As a precaution, the number of list elements and vector clocks that are expected to be found is compared to the number of actually found list elements and vector clocks. Should the number differ, an error message is printed.

3.4 Pre-Study

Initially, our setup used two CPUs. This decision was motivated by wanting to observe how often inconsistencies occur even under very restricted concurrency. We conducted the experiment for one and two threads with different load levels. For each combination, 20 memory dumps were created, and the execution time of LiME was not logged during the pre-study. Last, we proceeded with the analysis as described previously. LiME was loaded running `sudo insmod path/to/source/lime-4.15.0-177-generic.ko path=path/to/outputfile format=lime`.

Surprisingly, in many cases, we could not perform the analysis steps. Table 1 shows a summary of the encountered problems. Although we had created and tested a Volatility profile for the test environment, in some cases executing a Volatility plugin led to this error message: "No suitable address space mapping found." The heap extraction failed for even more memory dumps. A closer examination of the process list revealed that it was incomplete: processes known to have run while the memory was acquired were missing—among others, our pivot process. As Volatility uses the acquired process list to search the pid for which the memory range should be extracted, the heap of a process not found in the list cannot be extracted. In two cases, Volatility started the execution of the `linux_validate_vmas` plugin, but when trying to construct the list of tasks, an assertion error in one of the original Volatility scripts stopped the execution. We also observed that some of the memory dumps were smaller than the expected 4 GB.[3]

[3]We tried to investigate the reason for the memory dumps that were smaller than 4 GB after the main experiment was conducted. But even after having restored the system and experiment procedure (e.g., two CPUs, no logging of LiME's execution time, enabled LiME timeout), to the best of our knowledge, to their state at the time of the pre-study, dumps that were smaller than 4 GB were not produced anymore. However, the "no suitable address space" error did still occur.

To further investigate the problem, we reexamined the system load caused by the different load levels and concluded that it was difficult to slowly raise the CPU usage with only two CPUs and a multi-threaded pivot program. Therefore, we decided to change the number of CPUs to 4. We also enabled the debug output of LiME and created another 40 memory dumps. All were created with only one thread—half of them with medium and the other half with high system load. All 40 memory dumps had the expected size, but the “no suitable address space” error still occurred in 13 (respectively, 15) memory dumps out of 20. For both combinations, the pivot process’s heap could not be extracted from one more memory dump. This left us with 6 (respectively, 4) memory dumps for analysis.

A possible reason for this behavior was found in the debug output of LiME: LiME skips parts of page ranges when reading a page in the range takes more than 1 second. By default, this timeout is set to skip memory ranges for which reading is slow. Therefore, we repeated the same procedure loading LiME with timeout disabled. Within these newly generated dumps, we did not observe the “no suitable address space” problem, and only for three (respectively, six) memory dumps, no heap could be extracted. Therefore, we decided to continue the study with *four* CPUs and always loaded LiME with timeout *disabled*. Debugging was disabled.

3.5 Summary: A Hierarchical Classification of Analyzability

As a result of our experimental setup and the pre-test, we establish a hierarchical classification of memory dumps from the viewpoint of practical *analyzability*. This classification is based on the ability to perform certain steps of *structured analysis* in memory forensics and therefore assesses observable effects of inconsistency (without explaining their cause). The classification consists of the following four categories:

- *Broken*: This class comprises memory dumps for which not all memory was acquired (less than 4 GB) or are broken in other ways (e.g., or Volatility assertion error). In consequence, this class contains memory dumps that cannot be practically analyzed.
- *Meaningless*: This class comprises memory dumps which are complete and can be loaded into Volatility, but the process list only contains one process such that it does not make sense to analyze any further inconsistencies of virtual memory structures.
- *Valid*: The process list contains the pivot process, and process memory could be extracted.

Only valid memory dumps allow us to assess *causal consistency*. Note, however, that even a valid memory dump might still be causally inconsistent (even if the pivot program does not show any causal inconsistencies).

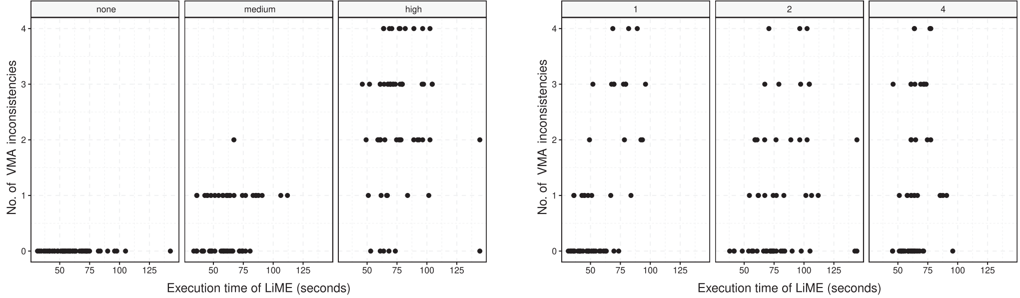
4 EXPERIMENTAL RESULTS

For the main experiment, the number of CPUs was set to 4 and all other settings were left untouched. The timeout of LiME was disabled by adding the parameter `timeout=0` when loading it with `insmod`. Furthermore, the pre-study revealed that some memory dumps were damaged (classification *broken*) or were missing information crucial to our analysis regarding inconsistencies (classification *meaningless*). To counteract, we decided to create 40 memory dumps for each parameter combination, ensuring 20 valid ones, which is our desired amount. Consequently, a total of 360 dumps were generated, analyzed, and classified (40 times 9 different combinations of parameters). Table 2 provides an overview of the classification of the created memory dumps for each load level. In contrast to the pre-study, no memory dumps were *broken*; all were either classified as valid or meaningless. The results of meaningless dumps regarding VMA and causal inconsistencies were removed from the overall results, and all other data (e.g., execution time of LiME, used memory) were stored separately. In case more than 20 samples per combination remained, we

Table 2. In the Main Experiment, 360 Memory Dumps Were Created in Total

Total	Broken	Meaningless	Valid ⊃ Final dataset	
360	0	102	258	180

Subtracting the meaningless memory dumps left us with 258 valid dumps, as defined in Section 3.5. To ensure the comparability of the results, for each combination of thread number and load level, 20 randomly selected valid memory dumps, 180 in total, were included in the final dataset.



(a) VMA inconsistencies in relation to the LiME execution time per system load level

(b) VMA inconsistencies in relation to the LiME execution time per thread number

Fig. 6. Relationship between time and VMA inconsistencies over all dumps.

Table 3. Number of VMA Inconsistencies and Number of Memory Dumps That Contained VMA Inconsistencies Per Load Level and Per Thread Number

		Number of threads						Total	
		1		2		4			
		Inc.	Dumps	Inc.	Dumps	Inc.	Dumps	Inc.	Dumps
System Load	none	0	0/20	0	0/20	0	0/20	0	0/60
	medium	6	6/20	10	9/20	11	11/20	27	26/60
	high	47	18/20	45	18/20	51	18/20	143	54/60
	Total	53	24/60	55	27/60	62	29/60	170	80/180

randomly deleted valid ones until 20 were reached. This was done to ensure comparability across the different settings. Thus, the final dataset comprises 180 functional dumps; 20 for each setting.

The following evaluation first presents the observations made on valid memory dumps, all dumps for which the heap of the pivot process could be extracted. Then, the observations made on meaningless dumps, the ones for which the heap extraction failed, are discussed.

4.1 VMA Inconsistencies

We observed no VMA inconsistencies for the *none* system load setting during the execution of LiME. However, as soon as system load was added, VMA inconsistencies appeared. Generally, they are more frequent for high system load than for medium system load. Figure 6(a) summarizes the spread of inconsistencies per load level and their occurrence in relation to the execution time of LiME. Table 3 shows the total number of VMA inconsistencies and how many memory dumps contained VMA inconsistencies per load level and per thread number. While under medium system load, slightly less than half of the created memory dumps contained VMA inconsistencies, and

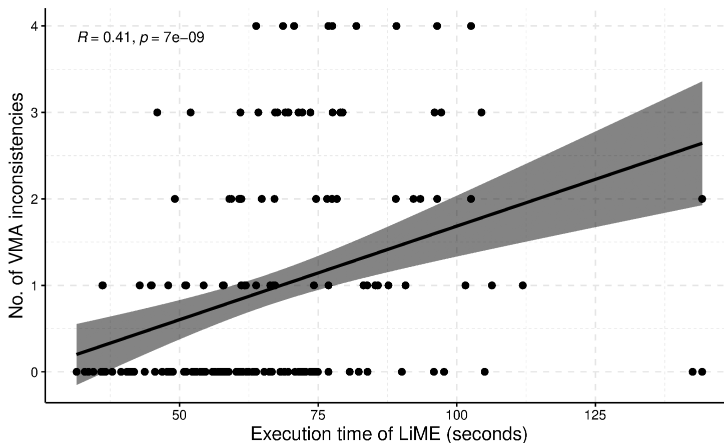


Fig. 7. Correlation between the execution time of LiME and the observed number of VMA inconsistencies.

Table 4. Mean and Standard Deviation (σ) of the LiME Execution Times in Seconds Calculated over 60 Memory Dumps for Each Load

System load	Mean exec time (s)	σ
none	60.88	19.11
medium	60.00	17.51
high	76.49	19.55

high system load caused inconsistencies in almost all the dumps. Additionally, under high system load, the highest observed number of VMA inconsistencies was 4, occurring in 10 memory dumps. Under medium system load, only one memory dump contained two VMA inconsistencies, and the other dumps contained at most one inconsistency. The number of threads does not seem to have a significant impact on the number of VMA inconsistencies, as depicted in Figure 6(b). The total number of inconsistencies per thread number, shown in Table 3, supports this. When more threads are running, the number of VMA inconsistencies as well as the number of memory dumps that contain them only rises slightly.

The Spearman correlation test indicates that there is a moderate monotonic relationship between the execution time of LiME and VMA inconsistencies. Figure 7 shows the Spearman correlation between the execution time of LiME and the number of observed VMA inconsistencies. The statistical relevancy of the observed moderate correlation is supported by the p-value, which is less than 0.05.

4.1.1 Load and Time. Given the observed correlation between time and VMA inconsistencies, as well as load and VMA inconsistencies, we take a closer look at the influence of the system load on the execution time of LiME. Figure 8 shows the distribution of the observed execution times of LiME separated by load level. Note that execution times are spread out widely for all load levels and contain outliers toward particularly long execution times. The standard deviations from the mean execution times, shown in Table 4, support this observation. The difference in execution times between none and medium load level is marginal. In fact, the mean execution time per load shown in Table 4 is less for medium system load than for none.

Figure 9 and Table 5 provide a detailed overview by splitting the observations according to the load level and the number of threads. Overall, we observe the highest execution times for two

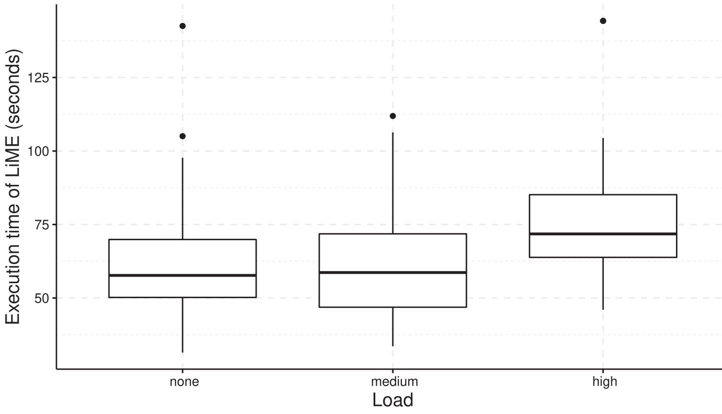


Fig. 8. Execution times of LiME grouped by load for all 180 memory dumps for which heap extraction was possible.

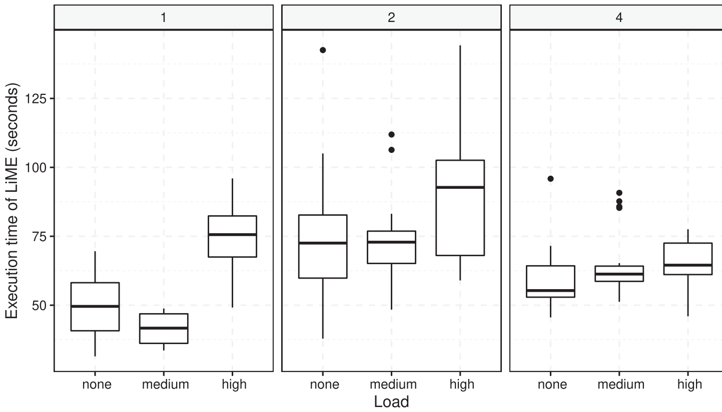
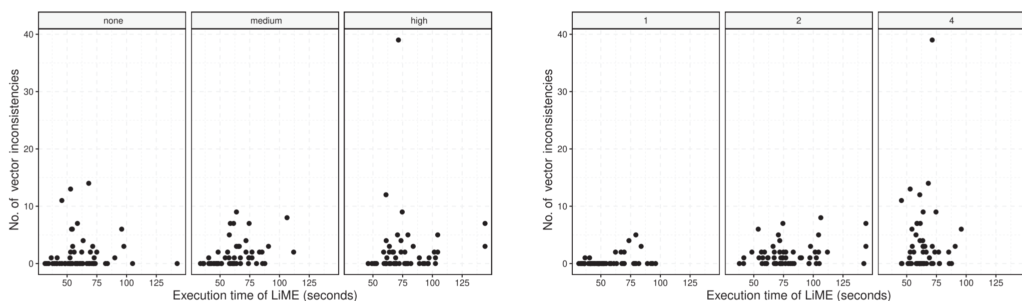


Fig. 9. Execution times of LiME per thread number grouped by load.

Table 5. Mean and Standard Deviation (σ) of the LiME Execution Times in Seconds Calculated over 20 Memory Dumps Per Load Level and Number of Threads

		Number of threads					
		1		2		4	
		Mean exec. time (s)	σ	Mean exec. time	σ	Mean exec. time	σ
System Load	none	49.61	11.08	73.49	24.06	59.53	11.37
	medium	41.90	4.98	72.78	15.44	65.33	11.92
	high	73.63	14.32	89.30	25.24	66.54	7.71

threads. But here the variance in execution times is also the highest for the different load levels as shown by the standard deviations. Whereas for one and two threads the highest execution times can be observed for medium system load, for four threads the mean execution time increases the higher the load level is. Additionally, the execution times for four threads were generally less varied than for the other thread numbers.



(a) Time to vector inconsistencies grouped by load level

(b) Time to vector inconsistencies grouped by thread number

Fig. 10. Relationship between time and vector inconsistencies over all dumps.

Table 6. Number of Causal Inconsistencies and Number of Memory Dumps That Contained Causal Inconsistencies Per Load Level and Per Thread Number

		Number of threads							
		1		2		4		Total	
		Inc.	Dumps	Inc.	Dumps	Inc.	Dumps	Inc.	Dumps
System Load	none	5	3/20	14	6/20	72	12/20	91	21/60
	medium	1	1/20	41	17/20	39	8/20	81	26/60
	high	14	4/20	32	12/20	76	10/20	122	26/60
	Total	20	8/60	87	35/60	187	30/60	294	73/180

When running the pivot program with two threads, three extreme outliers were observed. With no added system load, one acquisition took 143 seconds; with added system load, the acquisition took 124 seconds in two cases. In these cases, `stress` was not running during the complete acquisition, as it was only running for 120 seconds. All other runs were below 120 seconds.

4.2 Causal Inconsistencies

Causal inconsistencies were observed for all combinations of load levels and threads. For each dump, the expected number of list elements and vector clocks was found. Figure 10 shows both the distribution of inconsistencies found based on the vector clocks for the different load levels and the same distribution grouped by the number of threads in the pivot program. The thread number seems to have more impact on the number of causal inconsistencies than the load level. The relationship between causal inconsistencies, load level, and thread number is shown in Table 6. The number of inconsistencies rises more distinctly when the thread number is increased than when the system load is increased. The number of memory dumps that contain inconsistencies is also higher for two and four threads than for one thread. Comparing the number of memory dumps that contain causal inconsistencies across the different load levels shows only a small difference between load levels. This supports the impression that the number of active threads in the pivot process has a higher impact on the number of causal inconsistencies than the load. Note that for high system load and four threads, one extreme observation was made where 39 causal inconsistencies were counted in one memory dump while in all other memory dumps the highest observed number was 14.

To validate that the observed number of inconsistencies varies significantly for the thread numbers, a Kruskal-Wallis rank sum test was performed as the data does not seem to be normally

Table 7. Number of Meaningless Memory Dumps Per Load Level and Thread Number

		Number of threads			
		1	2	4	Total
System Load	none	17/40	9/40	16/40	42/120
	medium	16/40	11/40	8/40	35/120
	high	11/40	8/40	6/40	25/120
	Total	44/120	28/120	30/120	102/360

distributed. We assumed a significant observation for p-values smaller than 0.05. The test yielded $p = 1.556 \times 10^{-6}$, and therefore the differences between the observed numbers of causal inconsistencies for the different thread numbers were significant. A pairwise comparison using the Wilcoxon rank sum test with a Bonferroni correction was performed to identify between which thread numbers significant differences in the observed VMA inconsistencies exist. For the comparison between one and two threads, and between one and four threads, the computed p-values showed a significant difference between the groups. Between two and four threads, no significant difference was detected. The computed p-values can be found later in Table 13 in the appendix. In comparison, no significant difference between the observed causal consistencies across the different load levels was detected with the Kruskal-Wallis rank sum test ($p = 0.6858$).

4.3 Failed Heap Extractions

Table 7 shows the number of meaningless memory dumps per system load level and thread number. For these memory dumps, the heap of the pivot program could not be extracted. Overall, we encountered this more often with no added system load than with added system load. In total, 102 out of the 360 extractions failed. The lowest number of failed heap extractions was observed for high system load. The observed mean execution times of LiME are similar to those observed for the valid memory dumps. But it should be noted that for the failed heap extractions, the mean was not computed for each load level for the same number of samples.

In the output of the `linux_psxview` plugin for these memory dumps, the process list produced by the `linux_pslist` plugin contained at most one element which was always `systemd` (pid 1). Less often, the list did not even contain this process. The `linux_psscan` plugin found more task structures. However, often the structures of processes that are known to have been running during the acquisition, such as instances of `stress` or the pivot program, were not included. Sometimes, when the pivot process was found, the assigned pid matched that of the previous test run, not that of the current test run. In 15 cases, `linux_psxview` did not produce an output within 1 hour. However, running the `linux_pslist` and `linux_psscan` plugins separately did return output.

5 DISCUSSION

During the main experiment, data on the relation between system load and acquisition time, and different types of inconsistencies, was gathered. In the following sections, the observations are examined and possible explanations are discussed.

5.1 VMA Inconsistencies

The nature of the processes running concurrently with the acquisition has an influence on the likeliness to observe VMA inconsistencies. In the minimal execution environment chosen for our study, the only processes that often change their memory layout are those started by `stress`. Therefore, all observed VMA inconsistencies occurred in the task structures of the multiple

Table 8. Means of Used and Available Memory in Gigabytes for the Different Load Levels

System load	Mean used (GB)	σ	Mean available (GB)	σ
none	0.1206	0.0014	3.5	0
medium	0.3837	0.0363	3.2354	0.0359
high	0.6329	0.0382	2.9942	0.0379

instances of the program. One reason for observing more VMA inconsistencies under high load than under medium load may be that under high load, four instead of two processes are started that continuously perform `malloc` and `free`. However, this does not seem to be the only reason for the higher number, because although the number of processes performing the actions was doubled, more than five times the number of inconsistencies than under medium system load were observed.

Higher activity in the kernel under high system load could also contribute to the rise in inconsistencies. For high load, on top of four memory allocating processes, two are started that mostly demand CPU time. In total, four more processes are active under high than under medium system load. This should lead to more activity in the kernel, as the scheduler needs to frequently switch between those processes and background processes to assign system resources to all of them.

Another contributing factor could be the higher execution time for the high load level, observed in Section 4.1.1. If the acquisition takes longer for the kernel pages, changes in the memory layout of processes are more likely to take place between the acquisition of the pages. Therefore, changes to the different total number of VMAs, the corresponding structures, and the links between them happen while the pages are acquired. Thereby, the occurrence of VMA inconsistencies should become more likely. The results of Pagani et al. [31] support this assumption. This also matches the moderate correlation between longer execution times of LiME and a rise in VMA inconsistencies, shown in Figure 7. An exception can be seen in Figure 6(a) in the scatter plot for high system load. Here, for one of the extreme outliers of LiME's execution time, no VMA inconsistencies were observed. A possible explanation could be that during this run, the pages that took longer to acquire were located before or after those that contain the relevant kernel structures.

To further investigate the influence of higher load, we performed another iteration of the experimental procedure in which we increase the number of processes that are started with `stress`. Under high system load, four processes were started that allocate and free memory, and in the additional iteration, 16 are started. The number of threads is set to 4. For 1 of the 40 created memory dumps, the heap extraction failed. All of the 20 analyzed memory dumps contained VMA inconsistencies, and a total of 207 were found. Compared to the observed VMA inconsistencies for four threads and high system load shown in Table 3, the number of inconsistencies has increased about four times, the same factor with which the number of memory allocation changing processes was increased. We do not observe a comparably distinct increase in inconsistencies as when comparing the inconsistencies observed for medium and high load. The mean execution time of LiME is higher than for four threads, and load level high, it is 75.20 seconds ($\sigma = 9.68$).

5.1.1 Load and Time. An expectation was that when the load increases, the execution time also increases (i.e., similar to the behavior for four threads shown on the right in Figure 9). However, for one and two threads, we observed lower execution times of LiME for medium load than for no added load. In contrast, the used and available memory shown in Table 8 behaved as expected. With increasing load, the amount of used memory is increasing, and the amount of available memory is decreasing. The means were calculated for all successful runs based on the output of `free`

(see Section 3.2). The changes in available memory for no added system load were too small to be displayed by free with the chosen parameters. Therefore, a Kruskal-Wallis rank sum test was performed to verify if the differences in execution time across the different load levels for one and two threads are significant. We assume a significant observation for p-values smaller than 0.05. For the observations made for one thread, $p = 1.59 \times 10^{-8}$ is yielded, suggesting that the observed differences between at least two load levels are significant. The load levels for which the differences are statistically significant are identified with pairwise comparisons using the Wilcoxon rank sum test with Bonferroni correction. Between load levels of none and medium, no significant difference is detected. However, between load levels none and high, and medium and high, significant differences are detected. The computed p-values can be found later in Table 14 in the appendix. For two threads, no statistically significant differences between load levels are detected using the Kruskal-Wallis rank sum test ($p = 0.0929$). For the complete dataset, the p-value calculated with the Kruskal-Wallis rank sum test is 2.38×10^{-7}, and a significant difference in the observed execution times between at least two load levels exists. Further examination with a paired Wilcoxon rank sum test shows significant differences in the observed execution times between load level none and high, and medium and high. Between none and medium, no significant difference can be seen. The computed p-values can be found later in Table 15 in the appendix. In conclusion, even though, depending on the thread number, it looks like LiME's execution time is smaller for medium system load than none, no significant difference exists. Therefore, even though measurements with `top` show that under medium system load more time is spent executing processes than under system load none, the increased time the CPUs are spending executing other processes is not enough to increase the execution time of LiME on a statistically noticeable level.

5.2 Causal Inconsistencies

The observed relation between higher thread numbers and inconsistencies in the vector clocks is likely caused by the increased activity on the heap's pages during their acquisition. As changes to list elements are more frequent, it becomes likely that during the acquisition a change will be made that causes a causal inconsistency. The frequency with which changes occur could also be increased by decreasing the time the threads sleep after removing an element before reinserting it. Even if the sleep time was decreased, we assume that the increase of vector inconsistencies with increased thread number will at some point cease.

In contrast to the thread number, higher system load, as implemented in our study, increases the frequency with which changes in kernel data structures occur, such as changes in the number of VMAs a process allocated. It does not increase the concurrent activity on the pivot process's heap pages. However, in our minimal test setup, the heap's *physical* pages should not be highly fragmented. In a setup with more fragmentation, the observed prolonged execution time of LiME for the high system load level (see Section 4.1.1) could influence the occurrence of causal inconsistencies more distinctly. When it takes longer to acquire the heap's pages, it becomes more likely that new contents are written to already acquired pages before all pages can be copied. This increases the risk of missing causes of effects visible in the final memory dump.

5.3 Failed Heap Extractions

During the analysis of the output of the Volatility plugin `linux_pslist` for the memory dumps with which the heap extraction failed, the list often contained only the process with pid 1. Sometimes no processes were listed at all. The first case could be explained by the pointer from the task with pid 1 to the next task structure in the list being damaged. In the latter case, a damaged pointer from the `init_task` to the task with pid 1 could explain the observation. This could arise if the specific memory contents were acquired during an update of the pointers. But it seems improbable

that these pointers are updated very often. Therefore, the observation could point to inconsistencies in the dump causing problems for the analysis tool. Or, a bug in the analysis tool could cause the incomplete retrieval of the process list. Thus, for comparison, we examined the output of Volatility 3 [11], version 2.0.1, and Rekall [13], release 1.7.1, for the memory dumps on which the heap extraction failed and the `linux_psxview` plugin did not produce any output within an hour. Rekall is a memory forensic framework originally branched from Volatility. It is also Python based and provides its functionalities within various plugins. Because there are differences in the implementation of Volatility and Rekall, it seems meaningful to compare the outputs. However, since 2020, the project is not maintained anymore, and thus it was not chosen as the main tool for the evaluation. Volatility 3 is an updated rewrite of Volatility using Python 3. Some functionalities for Linux are still missing, and therefore it was not used as the main tool for the evaluation either.

In the presence of Kernel Address Space Layout Randomization (KASLR), one of the first steps for a meaningful analysis is to find out the virtual and physical shift to locate the kernel's address space. Without the shifts, the addresses of kernel symbols provided to Volatility with the profile cannot be correctly resolved. Therefore, we first compared the shift computed by each tool and then the reconstructed process lists. For Volatility, the plugin `linux_aslr_shift` was used to compute the shift. In Volatility3, we executed the plugin `linux.pslist.PsList` which extracts the process list with option `-vvv` for complete verbose output that includes the computed shift. Volatility and Volatility 3 produced identical virtual and physical shifts (it is possible that the algorithm for computing them has not changed between the versions). The Rekall plugin `find_kaslr` can return more than one possibility for the shifts which was the case for one of the examined memory dumps. It returned the same one as computed by Volatility/Volatility 3, and a different shift. Passing the first shift as an option to the `pslist` plugin did not produce more output. However, when passing the second shift, a longer process list was produced containing among others the pivot process and other processes known to have been running during the analysis. In this case, the right address for the PML4, called the *directory table base* in Rekall and Volatility, of the kernel could be found. For this memory dump, the implementation used to reconstruct the shift in Volatility and Volatility 3 seems to be unable to identify the right shift. For the other dumps, the same shifts were computed by Rekall as by Volatility and Volatility 3. For one memory dump, the plugin failed because only one possible PML4 address had been found but its verification failed. The process lists produced by Rekall are similar to those produced by Volatility. Sometimes a few more outputs are produced but they are unnamed, and all have pid 0. Here, an inconsistency in the memory dumps might hamper the correct computation of the shift and PML4 address. Or, the shift is computed correctly but inconsistencies, for example, in the paging data structures, do not allow to read the memory contents. It is also possible that all implementations contain an error that causes them to fail at identifying the right shift in these cases. With Volatility 3, no process list entries were produced at all for the 14 tested memory dumps. A requested page that could not be resolved caused the execution of the plugin to be aborted. The reported error was the same for all memory dumps.

With the `linux_psscan` plugin, sometimes task structures were found that looked like they were created for instances of the pivot program from the previous run. This is most likely possible because during a reboot the physical memory of the host system is not cleared. Even on a system that is not virtualized, this could happen because the RAM should be voltage carrying for most parts of the reboot. This does not pose a problem for our study, as we rely on the structures linked in the process list to find the heap of the pivot program. If it was an objective to evaluate the efficacy of carving algorithms, this would need to be considered when choosing an execution environment and designing the experimental procedure. Regarding the pivot program, it is possible that list elements from a previous run can also still be found in the memory dump. However, it is less

Table 9. Number of VMA Inconsistencies and Number of Memory Dumps That Contained VMA Inconsistencies Per Load Level and Per Thread Number for 8 GB RAM Size

		Number of threads					
		2		4		Total	
		Inc.	Dumps	Inc.	Dumps	Inc.	Dumps
System Load	medium	1	1/20	7	7/20	8	8/40
	high	0	0/20	10	6/20	10	6/40
	Total	1	1/40	17	13/40	18	14/80

Table 10. Mean and Standard Deviation (σ) of the LiME Execution Times in Seconds Calculated over 40 Memory Dumps for Medium and High Load

System load	Mean exec time (s)	σ
medium	120.06	21.10
high	152.51	36.83

likely that they are also contained in the newly set up process address space. Even if they were, this would not pose a problem, as the Python script used to recover list elements checks if the number of found vector clocks matches the expected number of list elements.

5.4 8 GB Memory Experiment

To gain an understanding of how well the experimental setup can be transferred to bigger RAM sizes, a part of the experiment was repeated for 8 GB. As for 4 GB, VMA inconsistencies appear only for medium and high system load. Consequently, additional experiments were only performed for these load levels. To reduce the number of experiments (and thereby the amount of data that had to be stored), the setting with only one thread running in the pivot program was excluded.

The observed VMA inconsistencies are shown in Table 9. Compared to the numbers for 4 GB shown in Table 3, fewer inconsistencies are observed. As in the experiments with 4 GB, the processes that are responsible for VMA inconsistencies are instances of `stress`. This could indicate that for bigger RAM sizes, more active processes are necessary to create memory pressure that affects the occurrence of inconsistencies. Another potential influencing factor can be identified by looking at the mean execution time of LiME for the bigger RAM size shown in Table 10. As the runtime of `stress` was not changed for the bigger RAM size and is still set to 120 seconds, at least in some cases its execution can be expected to have ended before the memory dump was complete. For medium system load in 13 cases, the memory dump creation took longer than 120 seconds, and for high system load, this was the case for 35 memory dumps.

Therefore, a second set of memory dumps was created for 8 GB. This time `stress` was executed for 180 seconds. All memory dumps were created with four active threads in the pivot program. For one set of memory dumps, the load level was set to high, and for the other, a higher load level called *high+* was used. For this load level, the number of running instances of `stress` that allocate memory is increased to 24. Table 11 shows the observed VMA and causal inconsistencies as well as the mean execution time of LiME for these cases. With the longer execution time of `stress`, the numbers of observed VMA inconsistencies match those observed for four threads and high system load with 4 GB of RAM. For the distinctly higher load level high+, a little bit more than twice the number of VMA inconsistencies than under high load are observed. This might show

Table 11. Number of VMA and Causal Inconsistencies, Number of Memory Dumps That Contained Them, Mean Execution Time, and Standard Deviation (σ) in Seconds, with an Increased Runtime of `stress`

		VMA incons.		Causal incons.		Execution time	
		No.	Dumps	No.	Dumps	Mean	σ
System Load	high	51	19/20	43	12/20	186.33	64.04
	high+	120	15/20	38	13/20	171.81	18.45

The threat number was always set to 4.

Table 12. Number of Causal Inconsistencies and Number of Memory Dumps That Contained Causal Inconsistencies Per Load Level and Per Thread Number for 8 GB RAM Size

		Number of threads					
		2		4		Total	
		Inc.	Dumps	Inc.	Dumps	Inc.	Dumps
System Load	medium	44	13/20	38	13/20	82	26/40
	high	21	7/20	31	12/20	52	19/40
	Total	65	20/40	69	25/40	134	45/80

the limits of increasing the number of running instances of `stress`, since six times more instances were running for load level high+ than for load level high.

Table 12 shows the number of causal inconsistencies observed for 8 GB of RAM. At first glance, the observed inconsistencies also seem to be less than those observed for 4 GB (see Table 6). However, for the smaller RAM size, one extreme outlier can be seen for four threads and high load (39 inconsistencies in one dump). Without this observation, the causal inconsistencies are in a similar range for both RAM sizes. Unlike the observed number of VMA inconsistencies, the occurrence of causal inconsistencies does not seem to be influenced by the shorter runtime of stress compared to the mean execution time of LiME. The observed causal inconsistencies for a longer runtime of stress which are shown in Table 11 are also in the same range. This supports the observation that causal inconsistencies are more dependent on the thread number than the load level.

5.5 Experimental Setup

The presented results were acquired with the setup described in Section 3. For the experiments, a VM in which Ubuntu without GUI was set up as OS was chosen as the execution environment. These choices were made deliberately to create an controllable environment. Using a VM allowed us to change settings such as RAM size and number of CPUs without having to use different hardware. Virtualization is often used in experiments regarding memory forensics (with the work by Pagani et al. [31] being a recent example). This is viable because the guest system is unaware of the virtualization and performs memory allocation as on a bare metal system. However, it should be noted that certain observations might be influenced by the virtualized environment, such as the difficulties with the `timeout` option of LiME.

To reduce unexpected peaks in memory usage and system activity, we chose an OS setup without GUI to minimize the number of active background processes. For the same reason, system load was only added by starting instances of `stress` on top of the pivot program. Thereby, variations between experiment iterations were meant to be reduced. With fewer variations, it is easier to see how the observed factors influence the occurrence of inconsistencies.

5.5.1 Pivot Program. Independent of the execution environment, the pivot program for which causal inconsistencies can be detected is a core element of the evaluation method. Using it has benefits but also limitations. With the pivot program, only the address space of one process is checked for causal inconsistencies. No parts of the kernel memory or other processes can be checked. However, if one process address space contains causal inconsistencies, it follows that the complete memory dump is not causally consistent. The size of the pivot program can also be adjusted to cover a larger range of memory, such as by starting it with more list elements. Alternatively, multiple instances of it could be used. By checking one process address space, it is also possible to observe the influence of fragmentation on the consistency within the heap of a process.

It would be beneficial to use the observation method for other processes as well, such as processes of interest to an investigation or in incident response scenarios. To do this, either adjustments would have to be made to the programs themselves or vector clocks would have to be managed at a higher level (i.e., from within the kernel). Similarly, checking the causal consistency of (parts of) the kernel address space could be insightful. Such an observation could be set up at the hypervisor level.

6 CONCLUSION AND FUTURE WORK

The objective of this study was to further the insights into the behavior of inconsistencies in main memory dumps under different conditions. To this end, we constructed an evaluation process, including a new method to observe causal consistency, and presented our evaluation results. The created memory dumps are made publicly available to support further research [14]. For our experiment, we chose a minimal execution environment to reduce uncontrolled influences on the system load and behavior. During the analysis of the memory dumps, we encountered expected inconsistencies but also unexpected problems like memory dumps that were smaller than the actual main memory.

Both the results of the pre-study and the inconsistencies observed during the main experiment underline the fragility of kernel level memory acquisition. Because of the likeliness of problems with memory dumps, depending on the circumstances, one countermeasure would be to acquire multiple memory dumps from a system under investigation. With each memory dump, information could be lost, but obtaining a memory dump that can be analyzed in a structured way is still beneficial because carving data structures is time consuming and error prone. Considering the correlation between the execution time and the number of VMA inconsistencies (see Section 4.1), in some cases it could be favorable to first obtain the memory of processes suspected to contain important information and then, if required, acquire a full memory dump, since inconsistencies in kernel structures could hinder the reconstruction of the address spaces of these processes.

Our observations in the pre-study also stress the importance of meticulously testing tools for the targeted systems and specific circumstances. The timeout feature added to LiME might be a helpful feature for some systems and RAM sizes, but in our case it led to more flawed memory dumps. If this happens during an investigation, the suspect system might not be accessible anymore when the problem is noticed (e.g., as a lot of time passed between acquisition and analysis and the relevant volatile data is not accessible anymore). The same is true for memory dumps smaller than the actual RAM size. Systematic evaluations are of course not only important for acquisition but also for analysis tools. One example presented in this article is the search for the cause of the failed heap extractions (see Section 5.3). Now that we know that there are (edge) cases for which Volatility cannot compute the address of the PML4 correctly, we can investigate this further and possibly improve the address reconstruction. Without a larger number of memory dumps created in the same environment, we might not have been capable of identifying this problem. All memory dumps for which the heap extraction failed during the additional experiments with 8 GB were kept

to provide a larger basis for further investigations about the reasons for the failures and possible improvements of analysis tools.

Many questions remain open for future work. Given the small RAM size and minimal system activity, our observations can be taken as a *lower bound* of possible inconsistencies. Repeating the experiment in an environment with more background noise, or with a larger RAM size like most PCs or servers have today, or on a smartphone, would allow us to study how the occurrence rate of the different types of inconsistency develops under more realistic conditions. It would also be interesting to know if there is a load level after which increasing the system load further does not increase the number of inconsistencies. Moreover, an experimental setup that increases the load on the system in a controlled manner but, unlike the experiment presented in this article, uses realistic user applications could give more insights. Especially, it could allow us to make a decision on when to acquire the memory of specific processes first.

Such a repetition of the experiments on a "normal" system, using real-world programs and a setup with GUI, instead of a minimal execution environment would also be interesting regarding the varying execution times of LiME we observed (Section 4.1.1) and the decrease of failed heap extractions with higher load (Section 4.3). If the observed effects are also visible when more processes are active, the influence of multi-threaded applications on the acquisition time could be investigated further. Since we observed high variances in the execution time, performing measurements with more than 20 memory dumps created per combination of load and thread number could be favorable to reduce the influence of acquisition time outliers on the mean acquisition time. When executing experiments in a non-virtualized environment, it would also be interesting to check if the `timeout` option of LiME is problematic in this setting as well, and if the variations in the execution times of LiME are similar to the ones reported in this article.

Overall, our results can also be interpreted as a call for acquisition approaches that achieve "more consistency." This was already observed by Pagani et al. [31], who suggested moving away from a purely sequential memory dumping approach. Instead, memory ranges known to contain vital information could be dumped first. Subsequently, all remaining parts of memory could be dumped. Thereby, a quick acquisition of the integral parts of the kernel memory could be ensured. Pagani et al. presented one possible implementation of the approach with a modified version of LiME that provides the option `smart`.[4] Such an approach should be included in future empirical evaluations.

Taking this approach further would be to more strongly integrate the sweep sequence of memory acquisition with activities of the OS. Similar to how memory is copied using the *fork* system call in Linux or in snapshots of main memory databases [25], memory could be acquired using a copy-on-write approach. Ottmann et al. [29] already conjectured that this would achieve *quasi-instantaneous* snapshots, which would be a quantum leap in kernel-based memory acquisition. Such advances could finally make *dirty* approaches achieve results that are similarly consistent with the *clean* approaches.

APPENDIX

A STATISTICAL TESTS

A.1 Variation of Observed Causal Inconsistencies

We verify if a significant difference between at least two groups exists using a Kruskal-Wallis rank sum test, as the data does not seem to follow a normal distribution. We assume a significant observation for p-values smaller than 0.05. The test yields Kruskal-Wallis chi-squared = 26.746, degrees

[4] https://github.com/pagabuc/atomicity_tops

of freedom = 2, and p-value = 1.556×10^{-6}. Therefore, a post-hoc analysis with the Wilcoxon rank sum test is performed, and Table 13 contains the computed p-values.

A.2 Variation of Observed Execution Times

We verify if a significant difference between at least two load levels for one thread exists using a Kruskal-Wallis rank sum test, and we assume a significant observation for p-values smaller than 0.05. The test yields Kruskal-Wallis chi-squared = 35.915, degrees of freedom = 2, and p-value = 1.59×10^{-8}. Therefore, a post-hoc analysis with the Wilcoxon rank sum test is performed, and Table 14 contains the computed p-values.

We verify if a significant difference between at least two load levels exists using a Kruskal-Wallis rank sum test, and we assume a significant observation for p-values smaller than 0.05. The test yields Kruskal-Wallis chi-squared = 35.915, degrees of freedom = 2, and p-value=1.59×10^{-8}. Therefore, a post-hoc analysis with the Wilcoxon rank sum test is performed, and Table 15 contains the computed p-values.

Table 13. Results of the Pairwise Comparison Using the Wilcoxon Rank Sum Test with Bonferroni Correction between the Numbers of Vector Inconsistencies across the Different Thread Numbers

Threads	1	2
2	$3.5x10^{-6}$	–
4	$1.6x10^{-5}$	1

Table 14. Results of the Pairwise Comparison Using the Wilcoxon Rank Sum Test with Bonferroni Correction between the Variations in the Observed Execution Times of LiME across the Different Load Levels for One Thread

Load	none	medium
medium	0.043	–
high	$4.8x10^{-5}$	$2.0x10^{-7}$

Table 15. Results of the Pairwise Comparison Using the Wilcoxon Rank Sum Test with Bonferroni Correction between the Variations in the Observed Execution Times of LiME across the Different Load Levels for All Thread Numbers

Load	none	medium
medium	1	–
high	$4.5x10^{-6}$	$6.5x10^{-6}$

ACKNOWLEDGMENTS

We thank Frank Block, Tobias Latzo, Fabio Pagani, and Florian Schmaus for their valuable feedback regarding our work.

REFERENCES

[1] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. 1993. Atomic snapshots of shared memory. *Journal of the ACM* 40, 4 (1993), 873–890. DOI: https://doi.org/10.1145/153724.153741

[2] Daniel P. Bovet and Marco Cesati. 2005. *Understanding the Linux Kernel: From I/O Ports to Process Management.* O'Reilly Media.

[3] William Campbell. 2013. Volatile memory acquisition tools—A comparison across taint and correctness. In *Proceedings of the 11th Australian Digital Forensics Conference.*

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

[5] Eoghan Casey. 2011. *Digital Evidence and Computer Crime: Forensic Science, Computers and the Internet* (3rd ed.). Academic Press. http://www.elsevierdirect.com/product.jsp?isbn=9780123742681

[6] Craig M. Chase and Vijay K. Garg. 1998. Detection of global predicates: Techniques and their limitations. *Distributed Computing* 11, 4 (1998), 191–201. DOI: https://doi.org/10.1007/s004460050049

[7] Chunbo Chu and Monica Brockmeyer. 2008. Predicate detection modality and semantics in three partially synchronous models. In *Proceedings of the 7th IEEE/ACIS International Conference on Computer and Information Science (ICIS'08).* IEEE, Los Alamitos, CA, 444–450. DOI: https://doi.org/10.1109/ICIS.2008.95

[8] Robert Cooper and Keith Marzullo. 1991. Consistent detection of global predicates. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging.* ACM, New York, NY, 167–174. DOI: https://doi.org/10.1145/122759.122774

[9] 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 (CODASPY'18).* ACM, New York, NY, 236–247. DOI: https://doi.org/10.1145/3176258.3176325

[10] Volatility Foundation. 2009. Volatility Framework—Volatile Memory Extraction Utility Framework. Retrieved October 30, 2023 from https://github.com/volatilityfoundation/volatility; Commit:a438e768194a9e05eb4d9ee9338b881c0fa25937.

[11] Volatility Foundation. 2019. Volatility 3: The Volatile Memory Extraction Framework. Retrieved October 30, 2023 from https://github.com/volatilityfoundation/volatility3; Commit:f8506862c4d92422a5e8927f70778f7faf69faf9.

[12] Felix C. Gärtner and Sven Kloppenburg. 2000. Consistent detection of global predicates under a weak fault assumption. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00).* IEEE, Los Alamitos, CA, 94–103. DOI: https://doi.org/10.1109/RELDI.2000.885397

[13] Google. 2013. Rekall Memory Forensic Framework. Retrieved October 30, 2023 from https://github.com/google/rekall; Commit:55d1925f2df9759a989b35271b4fa48fc54a1c86.

[14] Cinthya Grajeda, Frank Breitinger, and Ibrahim Baggili. 2017. Availability of datasets for digital forensics—And what is missing. *Digital Investigation* 22 (2017), S94–S105.

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

[16] Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* 12, 3 (1990), 463–492. DOI: https://doi.org/10.1145/78969.78972

[17] Phillip W. Hutto and Mustaque Ahamad. 1990. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *Proceedings of the 10th International Conference on Distributed Computing Systems (ICDCS'90).* IEEE, Los Alamitos, CA, 302–309. DOI: https://doi.org/10.1109/ICDCS.1990.89297

[18] Hajime Inoue, Frank Adelstein, and Robert A. Joyce. 2011. Visualization in testing a volatile memory forensic tool. *Digital Investigation* 8 (2011), S42–S51.

[19] Intel Corporation. 2016. *Intel©64 and IA-32 Architectures: Software Developer's Manual Volume 3A: System Programming Guide, Part 1.* Intel Corporation.

[20] Peter F. Klemperer, Hye Yoon Jeon, Bryan D. Payne, and James C. Hoe. 2020. High-performance memory snapshotting for real-time, consistent, hypervisor-based monitors. *IEEE Transactions on Dependable and Secure Computing* 17, 3 (2020), 518–535. DOI: https://doi.org/10.1109/TDSC.2018.2805904

[21] 504ENSICS Labs. 2014. LiME—Linux Memory Extractor. (2014). https://github.com/504ensicsLabs/LiME;Commit:9c734770f27f79e392cb726c97871bc27fbd013b.

[22] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 21, 7 (1978), 558–565. DOI: https://doi.org/10.1145/359545.359563

[23] Tobias Latzo, Ralph Palutke, and Felix Freiling. 2019. A universal taxonomy and survey of forensic memory acquisition techniques. *Digital Investigation* 28 (March2019), 56–69.

[24] Brett Lempereur, Madjid Merabti, and Qi Shi. 2012. Pypette: A platform for the evaluation of live digital forensics. *International Journal of Digital Crime and Forensics* 4, 4 (2012), 31–46.

[25] Liang Li, Guoren Wang, Gang Wu, Ye Yuan, Lei Chen, and Xiang Lian. 2021. A comparative study of consistent snapshot algorithms for main-memory database systems. *IEEE Transactions on Knowledge and Data Engineering* 33, 2 (2021), 316–330. DOI: https://doi.org/10.1109/TKDE.2019.2930987

[26] Michael Hale Ligh, Andrew Case, and Jamie Levy. 2014. *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*. Wiley. See also https://github.com/volatilityfoundation/volatility

[27] Friedemann Mattern. 1989. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*. 215–226.

[28] Jenny Ottmann. 2023. An Experimental Assessment of Inconsistencies in Memory Dumps: Experimental Data. Retrieved October 30, 2023 from https://doi.org/10.5281/zenodo.8246001

[29] Jenny Ottmann, Frank Breitinger, and Felix Freiling. 2022. Defining atomicity (and integrity) for snapshots of storage in forensic computing. In *Proceedings of the Digital Forensics Research Conference Europe (DFRWS EU'22)*.

[30] Jenny Ottmann, Üsame Cengiz, Frank Breitinger, and Felix Freiling. 2023. As if time had stopped—Checking memory dumps for quasi-instantaneous consistency. In *Proceedings of the Digital Forensics Research Conference USA (DFRWS USA'23)*. https://doi.org/10.48550/arXiv.2307.12060

[31] Fabio Pagani, Oleksii Fedorov, and Davide Balzarotti. 2019. Introducing the temporal dimension to memory forensics. *ACM Transactions on Privacy and Security* 22, 2 (2019), 1–21.

[32] Scott D. Stoller. 2000. Detecting global predicates in distributed systems with clocks. *Distributed Computing* 13, 2 (2000), 85–98. DOI: https://doi.org/10.1007/s004460050069

[33] Johannes Stüttgen and Michael Cohen. 2013. Anti-forensic resilient memory acquisition. *Digital Investigation* 10 (2013), S105–S115.

[34] Andrew Tanenbaum and Herbert Bos. 2015. *Modern Operating Systems*. Pearson Education.

[35] Ubuntu Man Pages. 2010. *Stress—Tool to Impose Load on and Stress Test Systems*, v1.0.4. Ubuntu.

[36] Stefan Vömel and Felix C. Freiling. 2011. A survey of main memory acquisition and analysis techniques for the Windows operating system. *Digital Investigation* 8, 1 (2011), 3–22. DOI: https://doi.org/10.1016/j.diin.2011.06.002

[37] Stefan Vömel and Felix C. Freiling. 2012. Correctness, atomicity, and integrity: Defining criteria for forensically-sound memory acquisition. *Digital Investigation* 9, 2 (2012), 125–137.

[38] Stefan Vömel and Johannes Stüttgen. 2013. An evaluation platform for forensic memory acquisition software. *Digital Investigation* 10 (2013), S30–S40.