Live System Call Trace Reconstruction on Linux

By:
Thanh Nguyen (Nvidia), Meni Orenbach (Nvidia), and Ahmad Atamli (Nvidia)

From the proceedings of

The Digital Forensic Research Conference

DFRWS USA 2022

July 11-14, 2022

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

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

DFRWS 2022 USA - Proceedings of the Twenty-Second Annual DFRWS USA

Live system call trace reconstruction on Linux

Thanh Nguyen [a, *], Meni Orenbach [b], Ahmad Atamli [c, d]

[a] NVIDIA Corporation, US 2788 San Tomas Expy, Santa Clara, CA, 95051, United States
[b] NVIDIA Israel Ltd, Yitzhak Sadeh St 4, Tel Aviv-Yafo, Israel
[c] NVIDIA Development UK, Milton Hall, Ely Rd, Milton, Cambridge, CB24 6WZ, United Kingdom
[d] University of Southampton, Southampton, SO17 1BJ, United Kingdom

ARTICLE INFO

Article history:

Keywords:
Memory forensics
Virtual machine introspection
System call tracing
Malware analysis
Ransomware

ABSTRACT

Live system call traces provide essential information in analyzing modern malware. Prior work demonstrated how system call traces can be used to differentiate benign from malicious applications. For example, ransomware invokes file system API to remove users' access to their sensitive data, and asks for a ransom to restore the access privileges.

Unfortunately, current methods and tools focus on offline reconstruction using memory dumps of the entire system. While it is possible to use such methods in live analysis by pausing execution and tracking system calls, it severely hinders the system performance. In this paper, we present the design and implementation of our method to trace system calls in Linux-based systems. We show how using our method enables obtaining and analyzing the system calls performed by real ransomware. To build a system call trace for a given process, we first retrieve the process context from a raw live memory image. Next, we analyze this context and identify the last invoked system call. We repeat this process generating a full trace.

We prototype our approach as a LibVMI tool using only memory read APIs and compare the accuracy of our method with that of strace, an intrusive and dedicated system call tracer. We show that we achieve the same level of accuracy, yet, without impacting the performance of the traced program. Further, we demonstrate how non-intrusive tracing has a better performance compared to trap-based tracing when using LibVMI traps and interrupts APIs to hook system call events. Finally, we provide an analysis of system calls performed by mature ransomware and demonstrate how our extracted traces can be used to identify them.

1. Introduction

Malware is continuously evolving and is notorious for its role in cyber−criminal activities. To protect against malware, virtual machine introspection (VMI) is a widely-used technique. VMI facilitates detection of malicious programs that have infiltrated virtual machines (VMs) through forensic analysis of volatile memory. For example, Volatility (The Volatility Foundation, 2021), Memflow (Memflow, 2022), and MemProcFS (Ulf Frisk, 2022) are frameworks that enable volatile memory analysis. Memory forensics is used to acquire information when incidents occur. It allows inferring the current state of a VM, such as the active processes, and loaded kernel modules. In turn, it is used to detect malicious activity in the VM. Disk forensics on the other hand aims to investigate the persistent artifacts left after cyber-security attacks take place. This paper focuses on memory forensics because it is a powerful tool for building the behavioral landscape of malware and advanced persistent threats by making it harder for malware to hide.

With the advancement of cyber-crime, detecting malware is becoming more challenging due to the malware's evolving ability to evade detection systems. Fortunately, behavior analysis through system call traces shows promise for distinguishing malicious from benign activity (Canali et al., 2012; Canfora et al., 2013, 2014, 2015; Isohara et al., 2011; Kolbitsch et al., 2009; Lanzi et al., 2010; Jeong et al., 2014; Reina et al., 2013; Schmidt et al., 2008; Tchakounté and Dayang, 2013; Wang et al., 2009). Furthermore, a system call trace provides a history of operations that the malware performed.

Prior work proposed several methods to trace system calls (Hebbal et al., 2015; Pham et al., 2014; Pfoh et al., 2011). First,

[*] Corresponding author.
E-mail addresses: thanhnguyen@nvidia.com (T. Nguyen), morenbach@nvidia.com (M. Orenbach), ahmada@nvidia.com (A. Atamli).

https://doi.org/10.1016/j.fsidi.2022.301398
2666-2817/© 2022 Published by Elsevier Ltd.

memory forensic methods that operate on a raw memory image. They use traditional techniques to infer executed system calls by analyzing kernel threads' stack state (Smulders, 2013). Second, trap-based methods that use the processors' ability to hook into events such as system calls and track them. Finally, utilities installed in VMs can trace system calls via the guest operating system (OS). For example, `strace` or eBPF-based tools such as Tracee (Aqua Security, 2022). However, this results in a non-negligible performance degradation (Zinke, 2009). Moreover, in-guest tracing tools result in an observer effect. Specifically, sophisticated malware can detect the tracing tools and render them ineffective. For example, the malware can obfuscate its behavior or disable the tracing tools altogether. Using VMI techniques places the data acquisition method outside the infected VM, which reduces malware ability to avoid detection.

Unfortunately, existing VMI methods are not a silver bullet: they are *intrusive* and impact the performance of programs executing in the VM. These methods are categorized as intrusive as they interfere with code execution. Trap-based methods force a VM exit and re-entry, resulting in a high latency penalty (Gordon et al., 2012). Furthermore, since existing memory forensic methods operate on a static raw memory image, collecting a live system call trace of processes executing in the VM mandates the following actions. Pausing the VM after each system call, acquiring the raw memory image, inferring the executed system call, and resuming the VM's execution. Thus, combining memory forensic techniques with the trap-based approach to trace live VMs also has a high impact on VMs performance and security analysts' responsiveness speed as we show in section Evaluation. Note, this performance degradation can be used by the malware itself to detect it is being monitored and attempt to hide its presence in the system. Therefore, existing intrusive VMI methods limit the applicability of system call tracing in production systems.

Non-intrusive system call tracing through VMI is a non-trivial task. A non-intrusive tracer must not interfere with VMs execution. Without special hardware support, non-intrusive tracing must rely on a separate thread, which actively traces programs for executed system calls. Yet, the tracing must be faster than the system calls. Otherwise, an executed system call would not be detected correctly as we discuss in section Background and Problem.

System calls are heavily used by programs, and current processors optimize their invocation latency. For example, the `getpid()` system call completes in 61 nano-seconds. Current methods that analyze an entire raw memory image do not complete in such a short time. Interestingly, we find that LibVMI's (Xiong et al., 2012) memory acquisition latency for a 4 KB page is much higher than existing system calls' latency. This makes it unsuitable for non-intrusive system call tracing. Furthermore, prior work (Pagani et al., 2019; Vömel and Felix, 2012) showed that it is not trivial to acquire a forensically-sound raw memory image. Specifically, if the acquisition method is not atomic the analysis results may be incorrect.

In this paper, we introduce the design and implementation of our method for a non-intrusive system call tracing on live VMs running the Linux x86_64 OS. Note, this OS is popularly used by enterprises and offered by major cloud vendors (Ama, 2021; Mic, 2021). To trace system calls for a particular process, a security analyst passes a process identifier to our tool. In response, the tool analyzes the VM's memory to recover registers data that the OS stores when system calls are invoked. Using the register values the tool reconstructs the system call trace.

We overcome the challenge of implementing a non-intrusive tracer that is faster than any Linux system call as follows. 1. We pinpoint the exact physical addresses of the registers used to recover the system calls by caching their virtual-to-physical translation in our tool. 2. Our tool communicates these addresses to a dedicated memory sampling framework we implement in the QEMU hypervisor. 3. We devise a novel method to use the same register values both for detecting system call execution and recovering system call information. We find this sampling technique to be fast enough to capture full system traces of widely-used applications as we discuss in section Evaluation.

We implement a prototype of our method on top of the popular VMI system LibVMI (Xiong et al., 2012). Our prototype includes non-intrusive system call tracing using the KVM-QEMU hypervisor Kivity et al. (2007); Bellard (2005), and an intrusive tracing utilizing the Xen hypervisor (Barham et al., 2003).

We confirm our method can accurately capture the same system call traces for applications as that produced by `strace`. We confirm this result both for benign applications and for malicious ransomware. Furthermore, we show the latency of the non-intrusive tracing is negligible, demonstrating its potential for malware detection using system call traces in production systems.

The contributions of this paper are as follows.

- We introduce a non-intrusive method for introspecting into system call traces on Linux-based VMs.
- We develop a non-intrusive extension to LibVMI-KVM that allows fast sampling of VMs' memory to trace system calls in live machines.
- We develop an intrusive VMI system call tracing based on LibVMI-Xen utilizes traps, which allows us to compare the performance impact of intrusive and non-intrusive tracing.
- We evaluate the intrusive and non-intrusive methods using popular off-the-shelf applications, and ransomware in live VMs. Our methods produce identical traces as `strace`, with the non-intrusive tool having negligible overhead compared to non-traced application execution.

To conclude, the construction of the paper is as follows. Section Background and Problem presents background information on system calls and the state-of-the-art VMI tracing methods. Section Approach Overview presents an overview of our tracing methods. Section Implementation Details discusses the implementation details as extensions on top of LibVMI. Section Evaluation presents the evaluation of the proposed method of tracing system calls using memory introspection and traps on benign applications and modern ransomware. Section Discussion provides a discussion on limitations and extensions for our method,. Section Conclusion concludes the paper.

2. Background and Problem

2.1. Background

System calls. In a typical execution of an OS kernel, such as Linux, application threads run in a physical core in unprivileged mode. Applications execute in this unprivileged mode until a trap handler transfers control back to the kernel in privileged mode. Invoking system calls involves the same process. Applications use special processor instructions, e.g. `int` 0x80 or `syscall` in the `x86` architecture, which transfer control to the trap handler that executes the requested system call. A simplified overview of system call invocation is depicted in Fig. 1.

For application developers, system calls act much like function calls, and often enough, function wrappers for system calls are provided by library developers to abstract OS-specific functionality. For example, `fread`() exists in `libc` and invokes the corresponding Windows and Linux system calls to read data from an open file

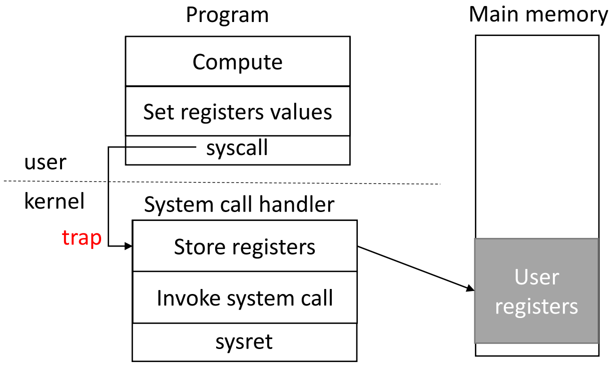


Fig. 1. Overview of system call invocation flow.

handler. Internally, to invoke the system calls, these wrappers set processor registers according to a well-defined calling convention, which both the OS and the processor adhere to. For example, in x86_64 Linux kernels, which is the focus of this paper, this convention is as follows. The `rax` register is used to indicate the system call number and store the system call result. The `rdx` register is used to signal an error, and finally the system call arguments are passed in `rdi`, `rsi`, `rdx`, `r10`, `r8`, and `r9` respectively.

The OS system call handler saves the user registers, invokes the requested system call, and sets the return value in the appropriate register. Before returning execution to the unprivileged program, the registers that are not defined as callee-clobbered are restored by the kernel.

Virtual machine introspection (VMI). VMI is a well-known technique that monitors the state of a virtual machine and performs forensic analysis on it. Traditionally, VMI utilizes the hypervisor, a thin layer of software that runs on top of the hardware to manage VMs. The hypervisor can access VMs memory, and apply traps, which enables information acquisition on the VMs' internal state. Finally, the hypervisor executes in a higher privilege mode than the VMs and is therefore not affected by processes executing in them.

Different frameworks exist to automate the process of VMI for different OSs such as Linux and Windows (Xiong et al., 2012; Garfinkel and Rosenblum, 2003). These frameworks facilitate the implementation of memory forensic tools that bridge the *semantic gap* and learn the behavior of processes executing within a VM. Prominent memory forensic frameworks include Volatility (The Volatility Foundation, 2021), and Rekall (Google Inc, 2019). Both are used to expedite and simplify forensic tools implementation as they provide high-level APIs.

Live analysis. Live analysis is essential for developing an endpoint intrusion detection system (IDS). IDS detects malware by continuously introspecting the status and the behavior of VMs. However, the approaches for live introspection vary in their level of intrusiveness.

High-intrusive methods rely on agents installed inside guest VMs to communicate with the introspection tools. However, agents are a prime location for malware to attack to avoid detection. Furthermore, agents' installation warrants change in the VMs, which is not applicable by all users.

Less intrusive methods utilize hypervisor control over VMs execution. For example, the hypervisor pauses the VMs via traps. Note, pause-less introspection would introduce *data races* (Pagani et al., 2019; Vömel and Felix, 2012). Specifically, introspection tools may read data that is being modified. Inconsistent data read may cause the introspection tools to fault or report incorrect results. On the other hand, VM pausing methods have a severe impact on the introspected applications' performance as shown in section Evaluation.

Finally, completely non-intrusive methods do not require VM changes, nor do they require pausing the VMs. These methods must overcome the data races and are usually used to track global VMs' states that are changed infrequently.

Using non-intrusive methods has multiple advantages. First, it removes the dependency on the hypervisor capability to pause/resume the VM. Second, it does not affect the execution of traced processes, which makes it an attractive option for live analysis in production systems and reduces detection by malware. Yet, non-intrusive methods have a limited visibility window into VMs execution compared to the intrusive methods. Thus, some behaviors cannot be inferred by non-intrusive methods. For example, a non-intrusive tool reading a memory region that is also updated by a VM process will observe stale data.

2.2. Live analysis of system calls

Given a raw live image of a VM, tracing system calls can be achieved by detecting whether a system call is currently executing, and recovering the CPU registers' values. The latter allows reconstructing the exact system call based on the system call calling convention. 4.

Detecting system calls with traps. Detecting system calls using traps is relatively straightforward for Linux (Pham et al., 2014; Pfoh et al., 2011). Effectively, the user places a trap on the system call handler address, `do_syscall_64`. The address is obtained from the OS intermediate symbol table (IST) generated from Linux's system map file. However, placing traps on every system call can cause a slowdown in the guest VM execution. We find that with this approach, a typical application that may take less than a second to run may now take more than a few seconds to run inside the VM. The slowdown is due to VM exits occurring on each trap, which is known to have high latency (Gordon et al., 2012) Indeed, we observe that the number of traps and respective slowdown correlates with the number of system calls an application invokes.

Non-intrusive system calls detection. System calls detection via live memory introspection is a non-trivial task, which is illustrated in Fig. 2. Essentially, application execution can be observed as a sequence of computations in unprivileged mode interleaved with system calls invocations. A non-intrusive introspection tool that polls for changes in the VM's memory to detect new system calls, whilst also inferring the executed system call must be faster than any system call invoked by the application. As seen in the figure, a fast introspection tool can poll and trace while the system call is being processed, whereas a slow introspection tool misidentify the system calls as the polling is too slow.

Modern processors are optimized to minimize system calls latency. Specifically, we measure the execution latency of the `getpid()` system call to be 61 nano-seconds. Note, `getpid()` is one the fastest system calls in Linux since its completion only depends

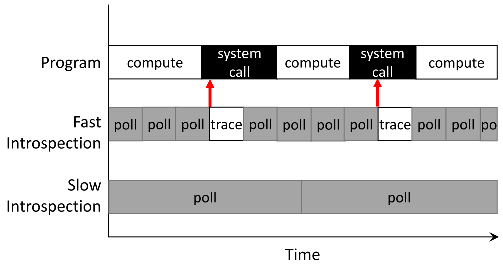


Fig. 2. Non-intrusive system call tracing challenges.

on the process identifier, which is stored in kernel memory. Thus, to correctly detect any system call, e.g., `getpid()`, a non-intrusive introspection tool must detect and infer system calls in less than 61 nano-seconds.

Recovering CPU registers. In a modern Linux OS, a mode switch from user space to kernel space involves the kernel's code storing a subset of the programs' registers in a dedicated kernel memory region as part of the kernel stack. Smulders (2013) reported the registers are stored as the first items on the kernel stack and pointed to by the `sp0` field, which is embedded in the `task_struct` structure's member `thread`. Note, `task_struct` is a structure containing per-process information to be managed by the Linux kernel. We find these registers' storage location is true for older kernel versions. Yet, this location changed in newer Linux kernels. Therefore, a new methodology is necessary for obtaining the values of the registers.

The goal of this paper is twofold. First, to provide a reproducible methodology to obtain stored user registers in the latest Linux kernels. Second, using the registers' values to recover the system call trace of VM processes.

3. Approach overview

This section provides a summary of our methods (see Fig. 3) to reconstruct system call traces from live VMs running a Linux OS. The reconstruction of system call traces is applicable for all generic hypervisors with access to volatile memory. Specifically, this includes the following steps.

- Obtain a context of a traced process thread.
- Infer the addresses of stored user registers for the context.
- Detect system call execution with intrusive and non-intrusive methods.
- Infer the system calls and constructs a trace file.

Next, we provide an overview of the intrusive and non-intrusive methods.

(a) Intrusive system call tracing method.
(b) Non-intrusive system call tracing method.

3.1. Intrusive tracing

In the Linux kernel, each system call goes through the same call gate: `do_syscall_64`. Therefore, in the preparation phase (left side, Fig. 3a), our tool places a trap on memory access to `do_syscall_64` physical address. Furthermore, the user sets a program to trace and the tool infers its kernel stack address, and the addresses of the stored user registers. Next, when the program executes and invokes a system call (right side, Fig. 3a), our tool analyses the register values and infers the invoked system call, appending it to the trace file.

3.2. Non-intrusive tracing

We find that the Linux kernel stores a unique number in the `rax` register field when system calls are invoked. Specifically, `-ENOSYS` is set before the system call begins and is modified once the system call concludes its execution. Thus, in the preparation phase, we set a polling thread (left side, Fig. 3b) to continuously monitor changes for this value. Next, similarly to the intrusive method, during the execution phase our tool analyses the values of the registers and infers the invoked system call (right side, Fig. 3b), and appends it to the trace file.

3.3. Tracing for malware analysis

System call analysis for malware identification gained popularity due to promising detection results. Behavioral models of malware were extensively studied. For example, prior work suggested using models based on system call sequences (Canzanese et al., 2015; Canfora et al., 2013, 2014, 2015). Both our tracing methods achieve perfect system call tracing accuracy, which enables using the recovered traces with such models.

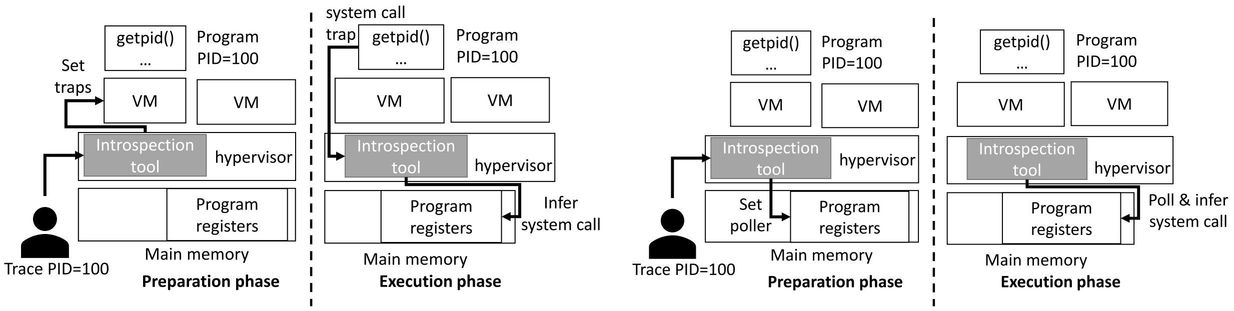
We note, however, that our tools currently do not produce the system call arguments as part of the traces or follow threads. While this may limit the usage of models that include the arguments (Canali et al., 2012), e.g., for incident response, it enables live analysis with negligible performance impact and reduced malware observation. We present the challenges and options for argument tracing in section Discussion.

4. Implementation details

Both methods are implemented in C and are built on top of LibVMI (Xiong et al., 2012). Furthermore, to support the non-intrusive method we extend the LibVMI interface with KVM-QEMU (Bit, 2021) with fast polling capabilities (see section Non-intrusive tracing). We test our implementation with both KVM and Xen, two popular hypervisors. We observe encouraging results as discussed in section Evaluation.

4.1. Preparation phase

In the preparation phase, our tool obtains information for the introspected application on the addresses where register values are stored upon system call invocation. Furthermore, our tool initializes a system call detection mechanism: traps for the intrusive method and polling for the non-intrusive method. **Obtaining thread**



(a) Intrusive system call tracing method.

(b) Non-intrusive system call tracing method.

Fig. 3. Overview of the system call tracing methods.

structure. Note that OSs generally store a thread context. For example, in Linux the task_struct structure contains per-process information. Extracting this structure's contents is well understood and can be achieved via existing implementations, such as the pslist plugin in the Volatility framework (The Volatility Foundation, 2021). We use the traditional LibVMI approach, which involves building a profile via the dwarf2json tool used by Volatility. The profile reflects the Intermediate Symbol Table (IST) for the Linux kernel and provides us with the explicit offsets of all the kernel symbols. In summary, the IST has the exact address of the process list head, and traversing the list utilizes LibVMI virtual-to-physical translation capabilities. The traversal stops when a task_struct with a user-provided process identifier is found. This task_struct is then used in the next phase for obtaining the values of this process's registers.

Obtaining the kernel stack. The task_struct structure contains information about a process, which includes its parent, children, name, virtual memory addresses and thread-related information as can be seen in Fig. 4. Our tool analyses the task_struct structure, which is stored in the kernel region of physical memory and follows the stack pointer, which points to the top of the corresponding thread's kernel stack. The kernel stack is used as a traditional stack when invoking functions in the kernel. Specifically, it stores stack frames matching function return addresses and arguments, as well as a frame pointer. However, the kernel stack also stores the thread_info structure at the top of the stack and the pt_regs structure at the bottom, which is followed by a magic value used to detect stack corruption attacks (see Fig. 4). The thread_info structure is used by the Linux kernel to compute the address of the respective task_struct structure, thereby enabling a two-way mapping between the kernel stack and the task_struct structure.

The kernel stack is page-aligned, and its size can be either 16 KB, or 32 KB. The actual value depends on whether KASAN was enabled when the kernel was compiled.

The IST does not contain the kernel stack size, which is required to compute the exact address of the pt_regs structure. Instead, we find two approaches that can be used to infer the stack size value. First, we observe that a kernel compiled with the KASAN feature contains related symbols in the IST, e.g., __kasan_kmalloc. Thus, our tool can infer the exact kernel stack size and use it to compute the pt_regs address. Second, we find that reading the memory of the kernel stack from the top of the stack will eventually reach an illegal address, which represents the stack bottom. We tested this on multiple Linux kernels, which demonstrated the same behavior. Finally, the top of the pt_regs structure can be obtained by subtracting the inferred address with the size of the pt_regs structure, which is available in the IST.

Obtaining the registers. Fig. 5 depicts the registers stored in the pt_regs structure. The IST contains the offsets for each register field relative to the start of the pt_regs structure. We use the address of pt_regs to compute the absolute address for each field. Thus, our tool can read the registers' values via the LibVMI's memory acquisition implementation.

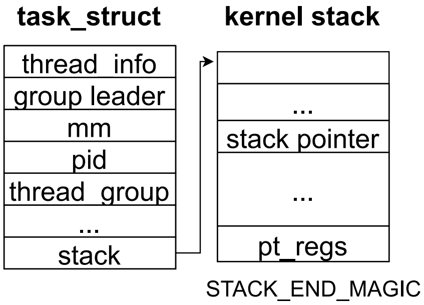


Fig. 4. Kernel stack held by task_struct

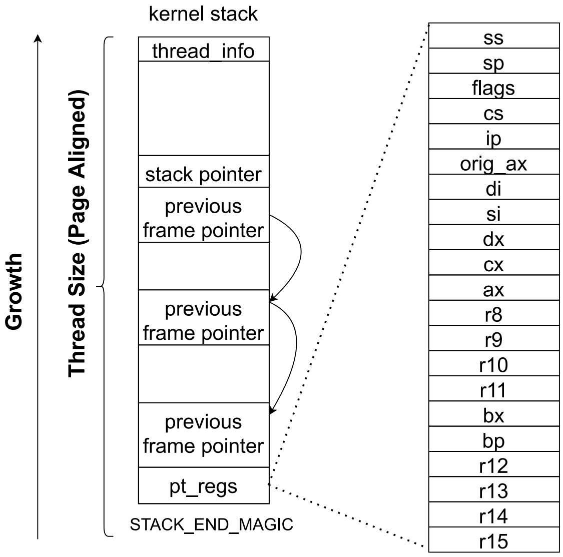


Fig. 5. Linux stores the registers at the bottom of the thread stack.

Obtaining the registers allows us to map the content of the registers back to a system call instruction. As described in section Background and Problem, in the x86_64 architecture the register rax stores the system call number that is being executed, which is used as an identifier to invoke the appropriate system call. The arguments for the system call functions are stored in rdi, rsi, rdx, r8, r10, and r9 in respective order.

Setting traps mechanism. The intrusive method is implemented by placing traps on memory access to do_syscall_64 physical address. The trap invokes a callback function that enumerates the calling process's memory by obtaining the per-thread structure. This is done in Xen by placing int3 that generates an interrupt on specific address access. When the address is accessed by the VM, the hypervisor invokes our tool's system call tracing before returning control back to the VM.

Setting polling mechanism. The non-intrusive method is implemented by polling accesses to the pt_regs structure to identify when the Linux kernel is processing a new system call. To that end, we use a separate thread that continuously polls the addresses of the registers to identify a new system call frame. We provide more details on the polling mechanism in section Non-intrusive tracing.

4.2. Execution phase

In the execution phase, our tool detects system calls and infers them.

Inferring system calls. The pt_regs structure contains both the ax and orig_ax fields. We observe that upon system call invocation the Linux kernel stores the system call number in the orig_ax field. Thus, our tool infers the system call identifier using the stored orig_ax value as an indexer to the Linux kernel system call table, which maps system calls to unique integer values.

Validating system calls. Invoking a system call instruction places the next program instruction in the rip register as the return address. Thus, the instruction before the return address should contain a valid system call instruction opcode. In the x86_64 architecture, available system call instructions include the SYSCALL, SYSENTER, and int 0x80 instructions, with a unique two-byte opcodes of 0x0F05, 0x0F34, and 0xCD80 respectively. To determine whether the values in the pt_regs structure represent a valid system call we read two bytes before the return address and match them with the aforementioned opcodes. Note, for the non-intrusive tracing this process is performed by a background thread, which hides the validation latency and does not compete with the tracing thread.

Finally, each inferred system call is appended to a trace file by our tool. Note, in Linux file writes are buffered in memory. Writing to the storage device is performed periodically by a background thread by flushing the buffered content. This enables non-intrusive tracing that maintains the timing requirements to detect all executed system calls.

4.3. Intrusive tracing

Putting it all together. All system calls force a callback in our tool, which reads the orig_ax value. With it, the tool infers the invoked system call, validates it and appends it to the trace file.

4.4. Non-intrusive tracing

Inefficient memory acquisition. We observe that the current memory acquisition implementation in LibVMI with KVM uses a Unix socket. The socket is used to pass requests to the hypervisor with physical memory addresses to read. In turn, the hypervisor reads the corresponding guest memory and returns the content as a response via the socket. Thus, analyzing the memory of a VM with LibVMI involves system calls invocation on the socket, i.e., writing, polling, and reading. As expected, we observe this memory acquisition implementation has a higher latency compared to some system calls executing in the VM, e.g., getpid(). Unsurprisingly, an earlier prototype we used to non-intrusively trace system calls with the existing memory acquisition mechanism failed to capture many system calls executing in the guest VM (as illustrated in Fig. 2 with the slow introspection tool). We conclude that the current memory acquisition implementation is not suitable for non-intrusive system call tracing.

To overcome this challenge, we implement a direct communication channel between LibVMI and the hypervisor, which is based on shared memory as illustrated in Fig. 6. Specifically, the hypervisor waits for requests on the shared memory region from the LibVMI tracing tool. Each request contains the pt_regs structure's physical address (pre-computed in the preparation phase), and each register offset, which is attained from the IST. Once a request arrives, a spinning thread is chosen from a thread pool to continuously poll the ax member in the pt_regs structure to detect a new system call frame as explained next. Once the application terminates, the thread returns to the pool and is free to handle new requests. Finally, since the polling thread continuously accesses this member, which fits in a single cache line, it enjoys the low latency offered by current processor caches. In fact, accessing a cache line is much faster compared to system calls' latency, which involves privilege mode crossings.

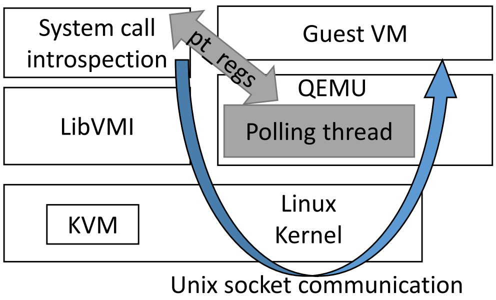


Fig. 6. Traditional LibVMI memory introspection is in blue, and our addition of the polling mechanism is in grey.

Note, in the preparation phase the pt_regs structure address is computed in the virtual address space. Polling for changes in the ax member requires to first translate the address to the corresponding physical one. Since the x86_64 architecture has a four-level page table, performing this translation on each access would exacerbate the polling latency, which in turn, would leave the non-intrusive tracing mechanism incapacitated. Thus, we use LibVMI's translation capabilities and pass the physical address to the polling thread.

Detecting new system call frames. Detecting a new system call through register introspection alone poses another challenge. Consider, for example, the following program reading a file 100 bytes at a time. For simplicity, we assume no errors are returned by the OS.

```
fd = open("file");
char buf[100];
while (nbytes > 0) {
    // read the file, 100 bytes at a time
    nbytes = read(fd, buf, 100);
    ...
}
```

Inside the loop's body, the same system call: read() is executed. Furthermore, the read() system call is invoked with the same arguments, and from the same virtual address. Therefore, the same register values describing this system call would be stored. Identifying that a new read() system call was executed using differential analysis on the stored registers' values is therefore impossible.

Transaction-based detection. To overcome this problem, we observe that in Linux, upon a system call the kernel stores the registers in the following order into the pt_regs structure: rip, rax, rdi, rsi, rdx, and rcx respectively. Interestingly, all the registers are stored to members representing their names. Yet, rax is stored in orig_ax. In addition, the pt_regs structure also contains a member named ax, which is set to the unique value -ENOSYS after all the aforementioned registers are stored. Finally, after the ax member is set, the kernel stores r8, r9, r10, and the rest of the pt_regs structure members.

After storing the registers' values, the kernel invokes the system call. Upon conclusion, the system call's return value replaces the -ENOSYS value stored in the ax member in the pt_regs structure. Thus, the ax member acts as a transaction cursor: it is set to -ENOSYS before a new system call frame is started and resets to a different value once it finishes. Note, to the best of our knowledge, existing system calls in Linux never return the -ENOSYS value. The polling thread thus monitors the ax member to infer when a system call is executing. Next, we discuss how the polling thread always observes the correct system call identifier in the orig_ax member.

The polling thread executes on a different core than the kernel. Since the x86 architecture employs the total store ordering (Sewell et al., 2010) memory model a store instruction is not reordered with respect to other store instructions. Thus, if the polling thread observes the -ENOSYS value in the ax field then a system call is currently executing. Since the kernel sets the orig_ax field prior to setting the ax field, then the orig_ax field must contain the correct value after observing the -ENOSYS value.

5. Evaluation

In this section, we confirm that our system call tracing method can accurately obtain system call traces of benign and malicious applications while performing intrusive and non-intrusive VMI.

We compare our extracted traces with the dedicated `strace` tool to validate the accuracy of the obtained results. Moreover, we measure and report the effect of system call tracing on applications performance and show that the non-intrusive VMI has negligible overhead and thus may be used to detect malicious activity in production systems. Setup. To validate the robustness of our method, we run the experiments on both AMD and Intel processors, and prototype it on top of both the Xen (Barham et al., 2003) and the KVM (Kivity et al., 2007) hypervisors.

First, we use an Intel Xeon Gold 6130 CPU with 64 GB of memory. The server utilizes LibVMI with Xen to execute the intrusive tracing experiments. Ubuntu 20.04.3 LTS was installed as the guest operating system.

Second, we use AMD EPYC 7352 CPU with 256 GB of memory. The server utilizes LibVMI with KVM to execute the non-intrusive tracing experiments. Ubuntu 20.04.3 LTS was installed as the guest operating system with Linux kernel 5.8.

Trace verification methodology. To verify that our generated system call traces are correct, we compare the sequence of system calls captured by our methods and that by `strace`. Specifically, we run the programs with `strace`, which generates a ground-truth system call trace file. Simultaneously, we apply our system call tracing via VMI and store our generated traces in a separate file. We validate both traces report the same set of system calls and in the same order. Furthermore, we validate the results are statistically significant by running each program 10 times and compare the trace files.

5.1. Benign applications

We evaluate several widely-used production applications, detailed in Table 1. These are popular and heavily used Linux programs that issue a plethora of different system calls. We consider this as a stable and diverse set of applications for us to compare our results with.

Intrusive method accuracy. The intrusive method generates a system call trace file by appending each inferred system call to the file. We observe that our method and `strace` generated traces are identical.

Non-intrusive method accuracy. Validating the non-intrusive method is challenging since polling for system calls must be initiated after each application begins its execution and before any system call is invoked. To that end, we observe that `strace` traces applications using the fork–exec paradigm. Therefore, we utilize Linux's `LD_PRELOAD` mechanism to hook the `fork()` system call. This lets us start our tool after the application is created via `fork()` and before it begins executing system calls.

Similarly to the intrusive method, we observe that for each application evaluated our method and `strace` generated traces are identical.

Table 1
Benign applications evaluated.

Program	Number of	Accuracy	Accuracy
	system calls	intrusive	non-intrusive
lspci	959	100%	100%
netstat	1,398	100%	100%
ps	2,299	100%	100%

5.2. Malware

Next, we present a system call trace generation of real malware with our methods. Specifically, we focus on ransomware: a type of malware that threatens to publish users data or perpetually block access to it unless a ransom is paid. We evaluate three open-source Linux ransomware, which are listed in Table 2. All ransomware are configured to work on a directory containing 100 files, each with 32 KB of randomly generated content.

RAASNet. RAASNet (Inc, 2021) is implemented in Python and is a type of ransomware as a service. It supports a wide range of features, such as generating a ransomware payload. It is available on Windows, macOS, and Linux, and provides a centralized server for administration.

Ransom0. Ransom0 (HugoLB0, 2021) is an open-source ransomware implemented in Python. It is designed to find and encrypt user data and decrypt it if a ransom is paid. The ransomware contains an HTTP server with an SQL database to store data such as keys used for encryption, digits, and the time of encryption.

Ransomware-POC. Ransomware-POC (jimmy ly00, 2021) is a simple ransomware proof of concept that is used alongside the open-sourced Atomic Red Team tool for testing the data encryption attacks. It uses an AES key to encrypt local files. The AES key is encrypted by an embedded RSA public key and sent to a command and control server after.

Tracing accuracy. We use the same setup and testing methodology as used for testing benign applications. Again, we observe that each of the ransomware evaluated has the same system call trace produced by both our non-intrusive and intrusive tools as the one produced by `strace`.

Latency Comparison. Measuring programs latency when executing in a VM is a non-trivial task. A hypervisor may emulate the time of the guests thereby hiding the actual execution time of programs on the underlying hardware (Tuzel et al., 2018).

For the non-intrusive method, we implement and execute a timing server in the hypervisor. The server waits for requests on a Unix socket to start/stop a timer on behalf of the requesting program. We use the server to measure the total execution latency of each ransomware evaluated.

To measure the ransomware latency with the intrusive method, we record the time when the first system call is invoked, and the time when the last system call is invoked. Both are trapped in the hypervisor, which measures the absolute execution time difference.

We run each program 100 times and measure the latency with and without our system call tracing tools. We report the median execution latency across all runs in Fig. 7. Note that the non-intrusive baseline is the speed of the native application in this case.

As expected, the non-intrusive method has negligible overhead for all the evaluated ransomware compared to the non-traced execution. However, the intrusive method suffers from high overheads due to the usage of traps. We observe that both RAASNet and Ransom0 have a slowdown of almost two orders of magnitude, whereas Ransomware-POC has a slowdown of about 15 × . This discrepancy is because the number of system calls in all three

Table 2
Ransomware applications evaluated.

Program	Number of	Accuracy	Accuracy
	system calls	intrusive	non-intrusive
RAASNet	9,694	100%	100%
Ransom0	13,811	100%	100%
Ransomware-POC	11,522	100%	100%

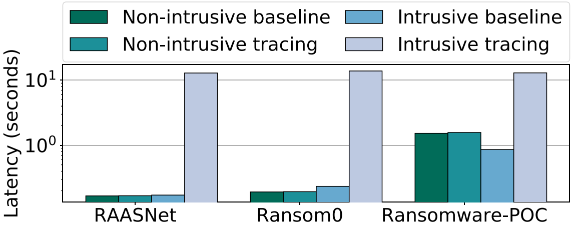


Fig. 7. Ransomware latency with/without out-of-band introspection.

ransomware is approximately the same. Yet, Ransomware-POC baseline latency is higher compared to RAASNet and Ransom0. Therefore, the contribution of the system calls latency and additional penalty due to traps in Ransomware-POC is smaller compared to the other ransomware.

System call traces to detect ransomware. Prior work showed that system calls traces can be used to identify malicious programs (Canali et al., 2012; Canfora et al., 2013, 2014, 2015; Isohara et al., 2011; Kolbitsch et al., 2009; Lanzi et al., 2010; Jeong et al., 2014; Reina et al., 2013; Schmidt et al., 2008; Tchakounté and Dayang, 2013; Wang et al., 2009). Next, we provide an analysis to show how the behavior of the ransomware correlates with their system call traces.

To that end, we first present the top 10 system calls in Fig. 8. As expected, we observe file system-related system calls. The reason is typical ransomware behavior which recursively scans directories for users' private files, reads them, and finally overwrites their content with an encrypted version.

Fig. 9 shows the sequence of invoked system calls for each of the evaluated ransomware. We categorize system calls based on their functionality. For example, `clock_gettime()` and `gettimeofday()` are both represented as Time system calls. The figure clearly shows the ransomware behavior of performing file operations throughout the ransomware lifetime in combination with memory management operation for buffer allocation and user data encryption. Interestingly, both RAASNet and Ransom0 use network-, time-, and synchronization-related system calls towards the end of their execution. The reason is that unlike Ransomware-POC they communicate with a server to transfer the victim's information and keys as a final step.

We conclude that with our non-intrusive method accurate system call traces can be obtained with negligible execution overhead and can be used to identify ransomware.

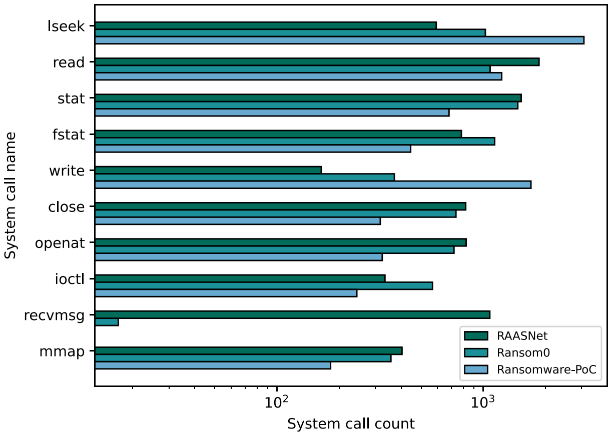


Fig. 8. Number of invocations for the ten most-used system calls.

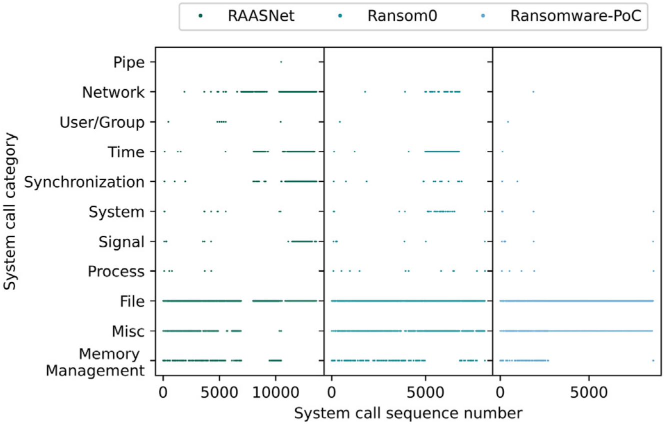


Fig. 9. Ransomware system call sequence grouped by category.

6. Discussion

Limitations. Our current prototype for the tool traces system calls, but not their arguments. This would be a useful enhancement, which may improve malware analysis in live VMs. To infer the content of the arguments, we need to implement a per-system call parser.

A limitation of the non-intrusive method is inferring the memory content of pointer-type arguments. For example, reading the content of the second argument of the `write()` system call: a buffer containing the to-be-written content. The reason is that dereferencing the pointer would be slow as it will require using the native LibVMI memory acquisition implementation, and require virtual address translation. Therefore, the tool may incorrectly parse the argument's data. Furthermore, current Linux kernels store the last three arguments (used by some of the system calls) after setting the `ax` field in the `pt_regs` structure, which we use to detect whether system calls are in-progress. Thus, the non-intrusive tool might read incorrect data for the last three arguments regardless of their type. Note, to fully capture arguments a hybrid approach combining intrusive and non-intrusive tracing can be used. Tracing system calls with arguments that cannot be captured non-intrusively will rely on traps. This will limit the performance impact of traps only to system calls with arguments that cannot be traced non-intrusively.

Our *current implementation* traces a single application thread. We plan to alleviate this limitation in future versions of the tool by supporting more tracing threads (one per application thread). However, security analysts can spawn multiple processes of our tools and trace different threads by providing their corresponding identifiers.

Disruption to our method. Interestingly, even polymorphic malware: a type of malware that constantly changes its identifiable features to evade detection cannot exclude system calls invocation to avoid detection. In fact, system calls represent services required by the malware to function correctly and are only serviced by the OS. However, malware that escalates its privilege to root access can override the system call handling procedures, and hide system calls from our tool. Note, changing the system call handling procedure is challenging as other features of the Linux kernel depend on it.

Applicability to other platforms. In this paper, we focused on tracing system calls in Linux for the `x86_64` architecture. However, the basic concepts introduced can be used for applying the method to other platforms. For example, the user register context is also saved into memory by the Windows OS (Otsuki et al., 2018) as Windows contains static pointers to the registers on its stack. To the best of our knowledge, for supporting the intrusive method a similar methodology can be used once we obtain the physical addresses for the stack's registers.

7. Related work

In-guest tools. `strace` is a system call tracing tool in Linux that intercepts and records systems calls based on `ptrace`. Tracee (Aqua Security, 2022) is a system-wide tracing tool based on eBPF. Both tracers output a sequence of system calls, their arguments, and return values. Unlike them, our method does not rely on in-guest tools.

VMI tools. Different VMI methods for stack tracing were proposed by prior work (Otsuki et al., 2018; Mahmood Hejazi et al., 2009; Arasteh and Debbabi, 2007).

Otsuki et al. (2018) present how to obtain stack traces from Windows `x64` memory dumps using the user-context of the targeted thread to locate the start of the stack, which allows for emulating stack unwinding and referencing the metadata for exception handling. In addition, a flow-based verification is proposed to scan for the stack trace in the case of malware manipulation with the forensic method proposed. Other work uses memory dumps (Mahmood Hejazi et al., 2009) to extract sensitive information such as passwords and usernames used by an application, while another (Arasteh and Debbabi, 2007) combined memory dumps with OS traps to obtain stack traces and construct a control flow graph.

Previous work on stack traces with Linux-based systems is outdated and does not apply to the latest versions of Linux (Smulders, 2019). Specifically, the location storing the registers' values was changed with the published method working for kernel version 2.6 and below.

Trap-based method to trace system calls was also widely explored by prior work. Ether (Dinaburg et al., 2008) leverages hardware virtualization extensions such as Intel VT-x, and AMD SVM to trace system calls using traps. Nitro (Pfoh et al., 2011) and HyperTap (Pham et al., 2014) are other VMI-based tools that trace system calls via traps. However, they provide a more portable solution that efficiently handles different system call instructions. HyperTap, Ether, and Nitro all have a higher performance impact compared to our non-intrusive method.

8. Conclusion

We introduced a non-intrusive method to trace system calls in live VMs running Linux. Our method obtains registers values stored by the kernel upon system calls' invocation for a traced process. Our method uses a novel polling mechanism to efficiently detect system call execution and infer each executed system call.

We evaluated the method on widely-used benign applications, and open-source ransomware and compared it to an intrusive method we implemented. We showed that both methods achieve the same system call trace as `strace`. Furthermore, we compared the latency of the two tracing methods to non-traced execution. The non-intrusive method had negligible overhead, whereas the intrusive method had a slowdown of two orders of magnitude.

The non-intrusive method demonstrates the potential for live malware analysis in production systems, e.g., for securing VMs operating in public clouds.

Acknowledgement

We thank the anonymous reviewers, our shepherd Erika Noerenberg, and Amir Naddaf for valuable feedback. This material is based on work supported by the European Union's Horizon 2020 research and innovation program under grant agreement No. 952697 (ASSURED).

References

Aqua Security, 2022. Linux runtime security and forensics using eBPF. https://github.com/aquasecurity/tracee.

Bellard, Fabrice, 2005. QEMU, a fast and portable dynamic translator. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC '05, vol. 41. USENIX Association, USA.

Canali, Davide, Lanzi, Andrea, Balzarotti, Davide, Kruegel, Christopher, Christodorescu, Mihai, Kirda, Engin, 2012. A quantitative study of accuracy in system call-based malware detection. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis. ISSTA, New York, NY, USA, ISBN 9781450314541, pp. 122–132. https://doi.org/10.1145/2338965.2336768, 2012. Association for Computing Machinery.

Canfora, Gerardo, Mercaldo, Francesco, Aaron Visaggio, Corrado, 2013. A classifier of malicious Android applications. In: 2013 International Conference on Availability. Reliability and Security, pp. 607–614. https://doi.org/10.1109/ARES.2013.80.

Canfora, Gerardo, Medvet, Eric, Mercaldo, Francesco, Aaron Visaggio, Corrado, 2014. Detection of malicious web pages using system calls sequences. In: Availability, Reliability, and Security in Information Systems. Springer International Publishing, ISBN 978-3-319-10975-6, pp. 226–238.

Canfora, Gerardo, Medvet, Eric, Mercaldo, Francesco, Aaron Visaggio, Corrado, 2015. Detecting Android malware using sequences of system calls. In: Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile, DeMobile 2015. Association for Computing Machinery, New York, NY, USA, ISBN 9781450338158, pp. 13–20. https://doi.org/10.1145/2804345.2804349.

Dinaburg, Artem, Royal, Paul, Sharif, Monirul, Lee, Wenke, 2008. Ether: malware analysis via hardware virtualization extensions. In: *Proceedings Of the 15th ACM Conference On Computer And Communications Security*, CCS '08. Association for Computing Machinery, New York, NY, USA, ISBN 9781595938107, pp. 51–62. https://doi.org/10.1145/1455770.1455779.

Garfinkel, Tal, Rosenblum, Mendel, et al., 2003. A virtual machine introspection based architecture for intrusion detection. In: Ndss, vol. 3. Citeseer, pp. 191–206.

Google Inc, 2019. Rekall memory forensic framework. http://www.rekall-forensic.com/.

Gordon, Abel, Amit, Nadav, Har'El, Nadav, Ben-Yehuda, Muli, Landau, Alex, Schuster, Assaf, Tsafrir, Dan, 2012. ELI: bare-metal performance for I/O virtualization. In: *Proceedings Of the Seventeenth International Conference On Architectural Support For Programming Languages And Operating Systems*, ASPLOS XVII. Association for Computing Machinery, New York, NY, USA, ISBN 9781450307598, pp. 411–422. https://doi.org/10.1145/2150976.2151020.

Hebbal, Yacine, Laniepce, Sylvie, Menaud, Jean-Marc, 2015. Virtual machine introspection: techniques and applications. In: 2015 10th International Conference on Availability. Reliability and Security, pp. 676–685. https://doi.org/10.1109/ARES.2015.43.

HugoLB0, 2021. Ransom0. https://github.com/HugoLB0/Ransom0.

Isohara, Takamasa, Takemori, Keisuke, Kubota, Ayumu, 2011. Kernel-based behavior analysis for Android malware detection. In: 2011 Seventh International Conference on Computational Intelligence and Security, pp. 1011–1015. https://doi.org/10.1109/CIS.2011.226.

Jeong, Youn-sik, Lee, Hwan-taek, Cho, Seong-je, 2014. Sangchul Han, and Minkyu Park. A kernel-based monitoring approach for analyzing malicious behavior on Android. In: *Proceedings Of the 29th Annual ACM Symposium On Applied Computing*, SAC '14. Association for Computing Machinery, New York, NY, USA, ISBN 9781450324694, pp. 1737–1738. https://doi.org/10.1145/2554850.2559915.

jimmy ly00, 2021. Ransomware-PoC. https://github.com/jimmy-ly00/Ransomware-PoC.

Pfoh, Jonas, Schneider, Christian, Eckert, Claudia, 2011. Nitro: hardware-based system call tracing for virtual machines. In: Advances in Information and Computer Security. Springer, Berlin, Heidelberg, ISBN 978-3-642-25141-2, pp. 96–112.

Kivity, Avi, Kamay, Yaniv, Laor, Dor, Lublin, Uri, Anthony, Liguori, 2007. kvm: the linux virtual machine monitor. In: Proceedings of the Linux Symposium, vol. 1. Dttawa, Dntorio, Canada, pp. 225–230.

Kolbitsch, Clemens, Comparetti, Paolo Milani, Kruegel, Christopher, Kirda, Engin, Zhou, Xiaoyong, Wang, XiaoFeng, 2009. Effective and efficient malware detection at the end host. In: *Proceedings Of the 18th Conference On USENIX Security Symposium*, SSYM'09. USENIX Association, USA, pp. 351–366.

KVM virtual machine introspection library. Bitdefender SRL. https://github.com/bitdefender/libkvmi/, 2021.

Lanzi, Andrea, Balzarotti, Davide, Kruegel, Christopher, Christodorescu, Mihai, Kirda, Engin, 2010. AccessMiner: using system-centric models for malware protection. In: *Proceedings Of the 17th ACM Conference On Computer And Communications Security*, CCS '10. Association for Computing Machinery, New York, NY, USA, ISBN 9781450302456, pp. 399–412. https://doi.org/10.1145/1866307.1866353.

Linux on AWS, 2021. Amazon Inc. https://aws.amazon.com/mp/linux/

Linux on Azure, 2021. Microsoft Inc. https://azure.microsoft.com/en-us/overview/

linux-on-azure/.

Mahmood Hejazi, Seyed, Talhi, Chamseddine, Debbabi, Mourad, 2009. Extraction of forensically sensitive information from windows physical memory. Digit. Invest. 6, S121−S131.

Memflow, 2022. - Machine Introspection Framework. Memflow. https://github.com/memflow/memflow.

Otsuki, Yuto, Kawakoya, Yuhei, Iwamura, Makoto, Miyoshi, Jun, Ohkubo, Kazuhiko, 2018. Building stack traces from memory dump of Windows x64. Digit. Invest. 24, S101−S110. https://doi.org/10.1016/j.diin.2018.01.013. ISSN 1742-2876. https://www.sciencedirect.com/science/article/pii/S1742287618300458.

Pagani, Fabio, Fedorov, Oleksii, Balzarotti, Davide, mar 2019. Introducing the temporal dimension to memory forensics. ACM Trans. Priv. Secur. 22 (2). https://doi.org/10.1145/3310355. ISSN 2471-2566.

Barham, Paul, Dragovic, Boris, Fraser, Keir, Hand, Steven, Harris, Tim, Ho, Alex, Neugebauer, Rolf, Pratt, Ian, Warfield, Andrew, 2003. Xen and the art of virtualization. ACM SIGOPS - Oper. Syst. Rev. 37 (5), 164−177.

Pham, Cuong, Estrada, Zachary, Cao, Phuong, Kalbarczyk, Zbigniew, Iyer, Ravishankar K., 2014. Reliability and security monitoring of virtual machines using hardware architectural invariants. In: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pp. 13−24. https://doi.org/10.1109/DSN.2014.19.

RAASNet, 2021. Incoming security. https://github.com/leonv024/RAASNet.

Canzanese, Raymond, Mancoridis, Spiros, Kam, Moshe, 2015. System call-based detection of malicious processes. In: 2015 IEEE International Conference on Software Quality. Reliability and Security, pp. 119−124. https://doi.org/10.1109/QRS.2015.26.

Reina, Alessandro, Fattori, Aristide, Cavallaro, Lorenzo, 2013. A System Call-Centric Analysis and Stimulation Technique to Automatically Reconstruct Android Malware Behaviors. ACM European Workshop on Systems Security (EuroSec).

Reza Arasteh, Ali, Debbabi, Mourad, 2007. Forensic memory analysis: from stack and code to execution history. Digit. Invest. 4, 114−125.

Schmidt, Aubrey-Derrick, Schmidt, Hans-Gunterh, Clausen, Jan, Ail Yuksel, Kamer, Osman, Kiraz, Camtepe, Seyit, Sahin, Albayrak, 2008. Enhancing security of linux-based android devices. In: 15th International Linux Kongress. https://eprints.qut.edu.au/58021/.

Sewell, Peter, Sarkar, Susmit, Scott, Owens, Nardelli, Francesco Zappa, Myreen, Magnus O., jul 2010. X86-TSO: a rigorous and useable programmer's model for X86 multiprocessors. Commun. ACM 53 (7), 89−97. https://doi.org/10.1145/1785414.1785443. ISSN 0001-0782.

Smulders, Edwin, 2013. User Space Memory Analysis. University of Twente. Master’s thesis.

Smulders, Edwin, 2019. Github−dutchy-/volatility-plugins: Container for Assorted Volatility Plugins.

Tchakounté, F., Dayang, P., 2013. System calls analysis of malwares on Android. Int. J. Sci. Technol. 2 (9), 669−674.

The Volatility Foundation, 2021. The volatility framework. https://www.volatilityfoundation.org/.

Tuzel, Tomasz, Bridgman, Mark, Zepf, Joshua, Lengyel, Tamas K., Temkin, K.J., 2018. Who watches the watcher? detecting hypervisor introspection from unprivileged guests. Digit. Invest. 26, S98−S106. https://doi.org/10.1016/j.diin.2018.04.015. ISSN 1742-2876. https://www.sciencedirect.com/science/article/pii/S1742287618301919.

Ulf Frisk, 2022. MemProcFS. In: https://github.com/ufrisk/MemProcFS.

Vömel, Stefan, Felix, C., 2012. Freiling. Correctness, atomicity, and integrity: defining criteria for forensically-sound memory acquisition. Digit. Invest. 9 (2), 125−137. https://doi.org/10.1016/j.diin.2012.04.005. ISSN 1742-2876. https://www.sciencedirect.com/science/article/pii/S1742287612000254.

Wang, Xinran, Jhi, Yoon-Chan, Zhu, Sencun, Liu, Peng, 2009. Detecting software theft via system call based birthmarks. In: 2009 Annual Computer Security Applications Conference, pp. 149−158. https://doi.org/10.1109/ACSAC.2009.24.

Xiong, Haiquan, Liu, Zhiyong, Xu, Weizhi, Jiao, Shuai, 2012. LibVMI: a library for bridging the semantic gap between guest os and vmm. In: 12th International Conference on Computer and Information Technology. IEEE, pp. 549−556.

Zinke, Jörg, 2009. System call tracing overhead. In: The International Linux System Technology Conference (Linux Kongress).