Agent-Based File Extraction Using Virtual Machine Introspection

Thomas Dangl(✉), Benjamin Taubmann, and Hans P. Reiser

University of Passau, Innstr. 43, 94032 Passau, Germany
thomas.dangl@uni-passau.de, {bt,hr}@sec.uni-passau.de

Abstract. Virtual machine introspection (VMI) can be defined as the external monitoring of virtual machines. In previous work, the importance of this technique for malware analysis and digital forensics has become apparent. However, in these domains the problem occurs that some information is not available in the main memory at all times. Specifically, files contained on non-volatile memory are typically not accessible for VMI applications. In this paper, we present a file extraction architecture that uses a dynamically injected in-guest agent to expose the file system for VMI-based analysis. To enable the execution of this in-guest agent, we also introduce a process injection mechanism for ELF binaries through the main memory using VMI.

Keywords: File extraction · Virtual machine introspection · Code injection

1 Introduction

Virtual machine introspection (VMI) is the process of monitoring virtual machines from the outside to gain knowledge of the inner state [7]. Due to this external monitoring of live systems, VMI has become an appealing technique for intrusion detection, malware analysis, virtual machine management, software debugging and memory forensics [11].

When dealing with *virtual machine introspection*-based malware analysis and computer forensics, many situations arise that require efficient access to non-volatile memory such as files that are stored on hard disk [15]. However, practical implementations for this use-case (when only access to main memory is given or the file system is encrypted) are lacking. In automated malware analysis, it is desirable to submit payloads that malware downloads to disk to the monitor for static analysis. For example, updates to malware should automatically be transferred to the monitor to track its evolution. For computer forensics purposes, it can be essential to obtain files contained in virtual machines during run-time without interruption of active services. Because files are typically not loaded into memory unless the user actively accesses them, performing forensics on the main memory is insufficient. Access to virtualized storage of the guest through virtual machine introspection instead of extracting the wanted data from the disk image

M. Asplund and S. Nadjm-Tehrani (Eds.): NordSec 2020, LNCS 12556, pp. 174–191, 2021.
https://doi.org/10.1007/978-3-030-70852-8_11

is required in situations where the target is protected by (full) disk encryption. Another reason could be that the targeted file is not stored on the VM itself, but is instead located on network storage, which is not accessible by the monitor.

In order to extract files from persistent storage, domain-specific solutions such as extracting credentials for *NFS* and *WebDAV*-based network storage or key extraction for encrypted volumes such as *encFS* or *LUKS* have been proposed [21]. However, those techniques only apply in their respective domain, as in many cases the file system type is not known in advance, is proprietary, or the technique relies on user actions.

In this paper, we design and implement a file extraction mechanism for use in VMI environments with the assistance of a dynamically injected in-guest agent that directly uses the file system capabilities of the guest. This proposed architecture is built with the following goals in mind: First, it must work on remote and encrypted file systems, this means the mechanism must operate independently of the underlying file system. Second, it should allow for reasonable transfer speeds so that the mechanism can be used to extract large files. Third, it must solely rely on existing introspection APIs without any modifications to the VMM. Last, it must be built considering stealthiness.

The contributions of this paper are the design, implementation, and evaluation of the following components that can be deployed in production environments on an unmodified Xen hypervisor using primitive VMI operations and events:

- A file extraction mechanism for files that are not loaded to main memory
- A process injection mechanism for VMI applications to execute ELF binaries
- A communication channel between an injected process and a VMI application

The outline of the paper is as follows: In Sect. 2 we present the common techniques of virtual machine introspection for hardware-assisted virtualization. The assumptions of our file extraction architecture and potential mitigation measures for the monitored virtual machine are discussed in Sect. 3. Section 4 introduces the components of the file extraction mechanism and outlines their interactions through VMI methods. In Sect. 5 we discuss the implementation of the VMI application and the in-guest agent that is injected into the monitored system as an ELF binary. Section 6 assesses the architecture based on transfer speed, performance degradation and stealthiness. In Sect. 7 we compare our work to the most related approaches concerning VMI-based code injection and file extraction. Finally, we conclude our findings in Sect. 8.

2 Virtual Machine Introspection

We begin by introducing the relevant terminology and the principles integral to the design and implementation of the file extraction architecture.

VMI and the Semantic Gap: Virtual machine introspection *(VMI)* was first designed to enhance robustness in intrusion detection systems by Garfinkel and Rosenblum in 2003 [7]. They defined VMI as the approach of inspecting a VM to analyze its behavior. Their first attempts at this novel technique involved a modified version of VMW are Workstation, which allowed the use of *direct memory access* (DMA) and access to virtual memory through manual address translation.

Pfoh et al. provide the theoretical foundation by describing a formal model for *virtual machine introspection* [24]. They still discuss this in the context of intrusion detection, but their results remain applicable for all VMI-based security applications. In particular, the research alludes to possible practical applications such as computer forensics and secure logging. One of the main issues identified here is the *semantic gap*, meaning the monitor requires assumptions over the internal state of the virtual machine, e.g., the memory layout, data structure layout, and kernel objects. The semantic gap is the problem of extracting high-level semantic information from low-level data sources [5].

Jain et al. summarize and compare multiple approaches concerning bridging the *semantic gap* [15]. They divide the problem of the *semantic gap* into sub-problems: The *weak semantic gap* refers to the challenge of creating *VMI-based* tools. The *strong semantic gap*, on the other hand, is the open problem of protecting such solutions from attacks interfering with the analysis, e.g., *Direct Kernel Object Manipulation*.

Furthermore, they categorize VMI-based monitoring of virtual machines as either *asynchronous* or *synchronous* [15]. Asynchronous monitoring refers to methods that perform analysis of RAM much like traditional memory forensic techniques, without manipulating the control flow inside the monitored VM. Synchronous monitoring on the other hand interferes with the control flow of the monitored VM, so that monitoring can take place at specific events or predetermined locations in the control flow, thus allowing a much greater level of control. This, however, requires support in the virtualizing hardware to perform context switches between VMs based on the monitored events.

Intel VT-x can perform a *VM-exit* when a software interrupt occurs within the guest virtual machine [14]. *VM-exit* refers to the event of a privileged instruction being executed, which traps to the hypervisor and executes the provided handler. This enables our code injection architecture to use the *int3* instruction to trigger a software breakpoint, which exits the virtual machine and allows the VMI application to intervene. Furthermore, we can perform a *VM-exit* on other events depending on the specific implementation. On Intel processors, this behavior can be configured using the primary and secondary *Processor-Based VM-Execution Controls*. Additionally, the monitoring of writes to certain control registers such as *CR3* is supported [14]. As this register acts as the default page table base register (*PTBR*), it must be updated by the scheduler when performing a context switch to an active process to reflect its page table, which enables synchronization through a *VM-exit* when a process becomes active within the monitored virtual machine. This makes synchronous VMI operations on specific

processes possible, which is required for many intrusive VMI operations such as code injection.

In this work, we use the Xen hypervisor and refer to the virtual machines using the terms introduced by Taubmann et al. [27]. The term *monitoring virtual machine* (MVM) is used for the virtual machine that performs the introspection and contains the VMI application. The MVM can either be the Dom0 or a DomU with the privilege to perform VMI operations on another VM. The *production virtual machine* (PVM) is the virtual machine that is monitored by the MVM.

VMI Tool Support: Bryan D. Payne [23] provides a library named *libvmi* based on *XenAccess*. This library aids in the prototyping of VMI applications. Through integrated support of existing *memory forensic* frameworks such as *Volatility* [30] and *Rekall* [25], bridging the *semantic gap* is significantly simpler in production environments as provisioning for different machines can now be automated. The bootstrapping of the in-guest agent via process injection and the resulting data transfer will heavily build upon this work.

Libvmtrace, a tracing library for virtual machines based on *libvmi*, is introduced by Taubmann et al. [28]. The library employs the previously mentioned technique of monitoring the *CR3* register to perform synchronous VMI operations. By doing so, the library can inject shellcode into an active process to perform *process forking* for the Linux operating system [26].

3 Threat Model and Assumptions

In this paper, we make the following assumptions regarding the system under analysis and discuss how a potential attacker that has access to the production virtual machine may potentially undermine our efforts. This aspect is relevant to the aforementioned use-cases when malware aims to prevent automated analysis or when the user of the virtual machine tries to impede an on-going forensic investigation.

First, we assume that the attacker does not compromise the kernel in a way that prevents the introspection from bridging the *semantic gap*. In particular, techniques such as *DKOM* are suitable to complicate or avert the use of virtual machine introspection for the use-case of file extraction [2]. Second, general kernel protection approaches such as (kernel) structure randomization [13] may prevent a successful application of introspection altogether. Given the case of randomization on the *task_struct*, the monitor would operate on the false assumption of a default data layout and would thereby be unable to correctly extract process information from the guest.

Furthermore, we assume that the attacker does not escape or bypass the isolation provided by the hypervisor and attack the file extraction VMI application directly. The hard disk's controller may not be modified or controlled by the attacker. Additionally, the guest virtual machine must allow for the execution of the covert in-guest agent. This means there must be no hypervisor or other mechanism in place that limits code execution on the guest by enforcing code

signing for all executables. Finally, the attacker can know about the presence of a hypervisor, but he may not be aware of the on-going introspection or code injection. Otherwise, it seems highly plausible to delete or hide sensitive files.

Moreover, the requested file must be accessible by a running process of the guest operating system. That is to say, the in-guest agent needs to be able to read the file after code injection. For this to be possible, it is expected that the kernel has not been modified, e.g., by placing hooks on relevant filesystem system calls. Also, the file system may not be monitored by relevant event-based callbacks in the kernel such as the *fanotify* API as this can be used to mitigate file access to relevant files. Lastly, the file system itself must not be compromised in a way that the relevant file can no longer be located by the PVM.

4 Methodology

In the following section, we describe the design of the system that is used to extract arbitrary files using virtual machine introspection. The following architecture is crafted with regards to the limitations of introspection APIs in off-the-shelf hypervisors, thereby enabling practical application in existing real-world systems.

As explained earlier, a typical guest OS supports many different kinds of file systems. Because the monitor might not know which file system to target and in the worst-case lacks the required implementation, we choose to directly use the file system capabilities of the guest, which makes our architecture suitable for general purpose file extraction by removing file system dependencies from the monitor.

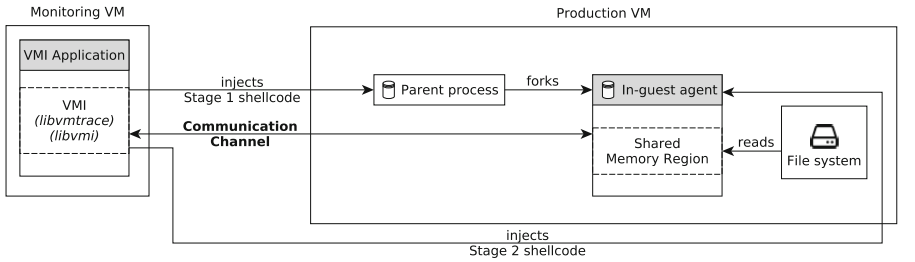

Fig. 1. Our file extraction architecture consists of a VMI application on the MVM and an in-guest agent injected on the PVM via a parent process. The in-guest agent reads the file system and transfers data to the MVM via a shared memory communication channel.

This shall be realized by injecting an agent into the guest system and establishing communication with this in-guest agent via shared main memory. Then, the in-guest agent exposes the file system tree and potential extraction targets through the communication channel, enabling file transfer across virtual machines.

4.1 Components

The primary aim of this paper is to extract files from the guest virtual machine. To achieve this goal, the file extraction architecture consists of two components. These components and their relationships are visualized in Fig. 1.

The first component of this architecture is the *VMI application*. It is executed in the monitoring virtual machine and performs the introspection of the PVM and is responsible for communication with the guest as well as for receiving the targeted file.

As previously mentioned, the extraction mechanism relies on an agent within the PVM. We consider this *in-guest agent* the second component of the file extraction architecture. The in-guest agent is bootstrapped by the VMI application using the technique described in the next section and provides the necessary insight into file systems available to the guest. Its purpose is to load a requested file into memory to make it accessible for VMI.

Because the file may be arbitrarily big, it is unfeasible to load the file into the main memory. Hence, the in-guest agent loads the requested file in chunks into memory. A *chunk* is one part of the file that fits inside the allocated buffer and can thereby be transmitted in one VMI operation. The shared memory region of the in-guest agent contains the file chunk and encodes relevant protocol data. In the context of VMI-based file extraction, shared memory refers to contiguous memory that is shared between VMs.

4.2 Procedure

An overview of the code injection and file extraction process is depicted in Fig. 2. Initially, we need to select a suitable user-mode process, which has the required

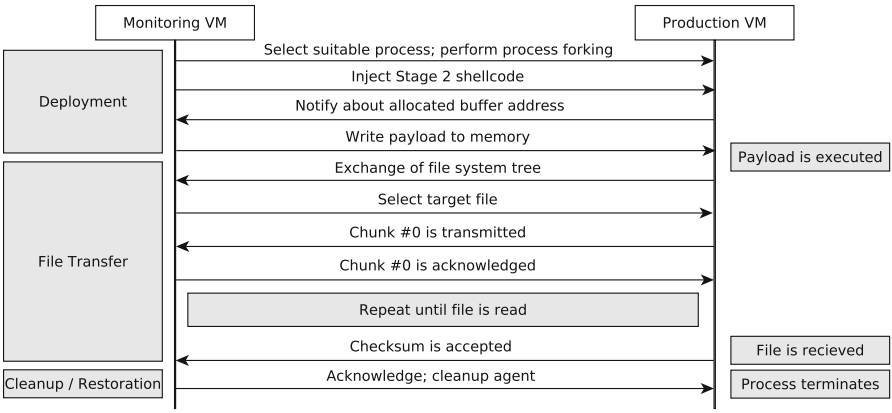

Fig. 2. The sequence of steps to accomplish file extraction consists of three phases: A deployment phase, the actual file transfer, and the restoration phase. The former two are performed using VMI-operations, while the latter is initiated by the agent after file transfer.

permissions to access the targeted file on the file system. We can identify suitable processes by extracting the file permission of the requested file from the file system and comparing them to the list of active processes as already implemented in *libvmi*.

After forking the selected process, the child process is replaced at run-time with the ELF executable in question. For this purpose, shellcode is injected into the child process. The monitor writes the ELF executable to a shared memory region created by the shellcode and resumes execution in the production virtual machine. Finally, the child process is replaced by the ELF executable transmitted from the monitor.

After the in-guest agent has been deployed in the guest machine, the actual process of file extraction begins. As the user of the VMI application possibly does not know where files of interest are stored within the guest virtual machine, the extraction mechanism enables the VMI application to query a full file system tree of the guest virtual machine. The result of this operation is transmitted to the MVM by the same mechanism that is used to transfer the targeted file. This enables the application to query and pick an arbitrary file present on the guest without prior configuration of the in-guest agent.

Once the target has been selected and requested by the VMI application, the in-guest agent determines the size of the file. At this stage, the guest reads the current chunk into memory and signals the monitor that the buffer is clear to read. When the respective chunk has been received by the application, it is stored off to a file on the MVM. Now the only thing left to do is for the VMI application to notify the in-guest agent that it may begin to transfer the next *chunk*. This process repeats until the entire file has been transmitted to the monitor.

4.3 Communication Channel

To establish communication between MVM and PVM, the in-guest agent exposes an interface through shared memory. This interface allows the VMI application to request files and the file system tree. It is also used to transfer the file to the MVM. How the application can interact with this memory region is elaborated on in Sect. 5.2.

This communication channel behaves like shared memory as supported by many operating systems such as *Linux* and *Microsoft Windows NT*. When transferring the targeted file to the monitor via VMI, a file chunk may only be unmapped and replaced by the next chunk when the VMI application has already stored off this particular chunk. In common non-VMI applications such synchronization would be provided by techniques such as *mutexes* or *semaphores*, typically implementations for these procedures are supplied by the operating system. However, this means we cannot rely on them to guarantee mutual exclusion as this makes them unsuitable for use across virtual machines.

Instead of reimplementing these mechanisms for VMI use, we provide synchronization in the presented file extraction architecture through a *spinlock*,

which can easily operate independently from both operating systems in the MVM/PVM and is simple to implement.

5 Implementation

This section deals with the implementation of both architectures according to the design principles outlined in the previous Section. All of the following implementations are built upon the *libvmi* introspection library. Because the task of file extraction requires access to kernel structures to perform the code injection and to find relevant user-mode task information, a *Rekall*-profile is necessary to obtain the structure offsets [25]. We implemented our solution for Linux 4.5 and believe that adaptions for other operating systems are possible.

5.1 Code Injection

In the following the characteristics of the code injection procedure for ELF executables across virtual machines are elaborated. The goal of this technique is to inject arbitrary user-mode programs into virtual machines solely through main memory. As alluded to earlier, this process will be performed in two stages: First, an eligible process with suitable rights for file system access must be forked using VMI. Then the child of this process fork is replaced at run-time with the designated ELF executable.

Figure 3 shows the sequence of actions taken without consideration for synchronization. To perform the first step of this procedure—*process forking*—the host must know when and where the vCPU is executing code in the forked user-mode process. For this purpose, we monitor changes to the *CR3*-register, where a pointer to the top-level paging structure is held. By doing so, we can perform synchronous VMI operations when the guest OS scheduler switches to our targeted process. To determine the address at which the program execution will continue, we read the future instruction pointer directly from an offset to the kernel-mode stack pointer. After we injected the *stage 1 shellcode* at this location *(1)*, it will first perform a *vfork* in the parent *(2 & 3)* and then a *execve* in the child. When the *execve* system call in the shellcode is reached, we use the VMI application to store the system call arguments under the user-mode stack pointer. Additionally, the *stage 1 shellcode* must preserve the registers *RAX*, *RCX* and *R11* in the parent process as these are modified by performing a system call.

For the use case of file extraction, the *execve* system call will execute */bin/bash* within the child process, thereby putting it into an infinite loop, which causes frequent context switches to the target process by the Linux scheduler [20]. To inject the ELF executable into this newly created child process in stage 2, we once again employ the technique of monitoring changes to the *CR3*-register to synchronize with the guest system. However, in the use-case of the proposed ELF injection, it must also be taken into consideration that the forked child process might still use the parent's page tables when a CR3 event is first triggered [8].

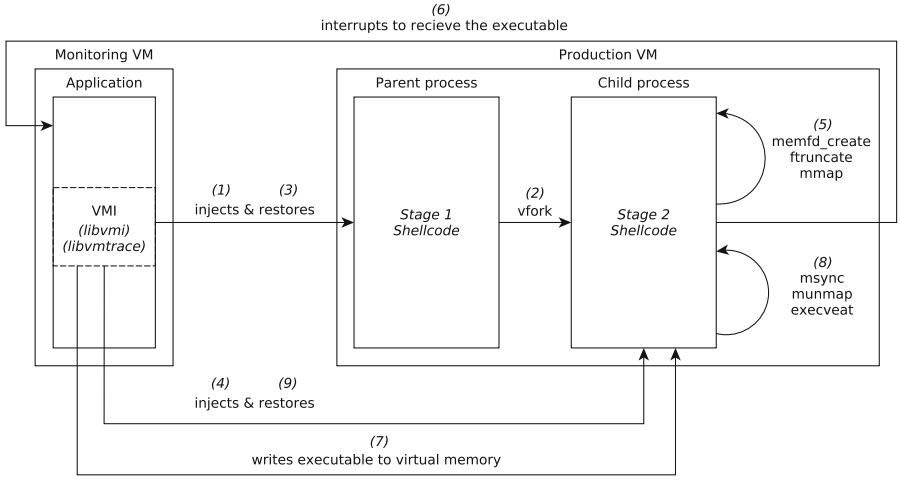

Fig. 3. Our VMI-based ELF injection implementation first forks a process using code injection. Then, we replace the child of this fork at run-time with an executable transmitted by the host.

Because the operating system does not duplicate the page tables of the child process when using *vfork*, both the parent and the child process can refer to the same top-level paging structure until *execveat* is called, thus not allowing any distinction between them. To deal with this issue, it must be ensured that the code injection is delayed until the above procedure is completed[1]. Eventually, we can continue the injection of the *stage 2 shellcode* at the future instruction pointer *(4)*.

In stage 2, initially, a file descriptor to an anonymous file[2] is opened by the shellcode *(5)* using the *memfd_create* system call. This is required because the Linux operating system can execute programs only from files. To reduce the chance of detection, the *MFD_CLOEXEC* flag is used, so that the descriptor closes on program execution. Afterward, the entire file is mapped to virtual memory using the *mmap* system call *(5)*.

At this point the shellcode performs a context switch to the VMI application *(6)*. Now there are two things that must be taken care of: First, the in-guest agent must be written to the buffer *(7)*. Second, measures must be taken to restore the previously backed up memory region that was overwritten by the

[1] This is achieved by waiting in the VMI application until the child's top-level paging structure differs from the parent's.

[2] Under Linux operating systems, the term *anonymous file* refers to a file that lives solely in memory. It is not present on any mounted file system and released once it is no longer referenced [12]. The *memfd_create* system call was introduced in version 3.17. For older Linux versions or BSD variants, it is possible to use *shm_open* instead.

shellcode. For this purpose, a breakpoint is placed in kernel-space at *LSTAR*[3] before the *execveat* system call is handled.

Subsequently the shellcode synchronizes the now mapped ELF executable to the file descriptor and cleans up the allocated memory used for the transfer. To finish the injection, *execveat* is invoked with the file descriptor, which discards the anonymous file and replaces the current process with the provided program *(8)*. The only thing remaining is to restore the original instructions from the monitor when the previously placed *execveat*-breakpoint is executed *(9)*.

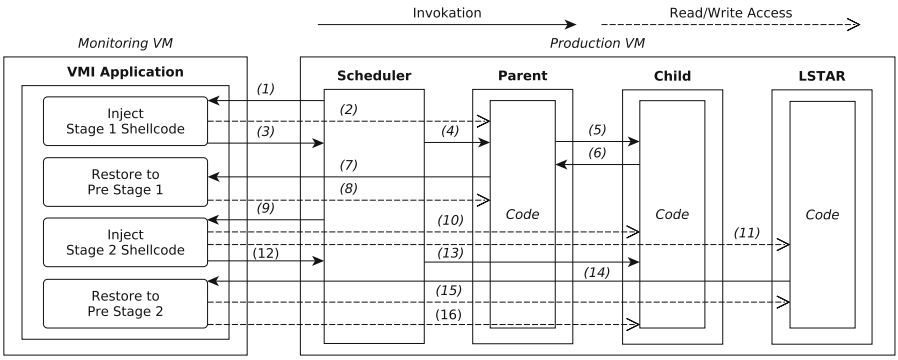

Fig. 4. During the ELF injection, multiple VMX context switches and VMI-based write operations occur between VMI application, scheduler, parent and child process, and system call handlers.

Figure 4 depicts essentially the same process as shown in Fig. 3, however, in this instance, we consider context switches and read/write operations between MVM and PVM instead of control flow. While this architecture for injecting ELF binaries is in theory applicable to any hypervisor, lacking support for event handling in *libvmi* for other hypervisors currently limits the practical applicability to *XEN*.

5.2 File Extraction

After the groundwork has been laid, the details of the file extraction implementation are discussed. As described in Sect. 4.2, the previously introduced code injection mechanism is used to deploy the in-guest agent within the PVM. This in-guest agent will perform all file system related operations and aid in extracting the file. For communication purposes, the in-guest agent exposes a shared memory region as a symbol through its ELF export directory, which can be located in virtual memory by the VMI application.

[3] *LSTAR* is a model-specific register that holds the targeted instruction pointer when executing a system call in long mode.

Before any communication is established, the in-guest agent allocates the transmission buffer on the heap. The size and location of this transmission buffer is written to the shared memory region, so that the VMI application knows how many bytes it can read. To prevent the PVM's operating system from paging out the buffer, we lock it into virtual memory using the *mlock* system call for the duration of the in-guest agent's execution.

If the user wishes to skip the transfer of the file system tree, the VMI application signals this decision to the in-guest agent via a bit-flag in the shared memory before any other operation takes place. Otherwise, the in-guest agent pipes the result of *tree*/into an anonymous file, which is then transmitted by the same mechanism as explained below.

After the user has decided on which file to extract, the respective file path is written to the transmission buffer. Before the transmission begins, a *CRC-32* checksum of the entire file and the total file size is stored within the shared memory. Now, the transmission may begin and the targeted file is read chunk by chunk into the transmission buffer by the in-guest agent. After each step, the agent uses a bit-flag to indicate the buffer contents are valid again. As the VMI application is pulling on this specific bit, spinlock alike behavior ensures. On each successful pull, the VMI application extracts the current file chunk and stores it off. In order to signal that the chunk was received correctly, the application flips bit-flag again. If this chunk transfer completes the entire process, the in-guest agent terminates in order to evade potential detection after file extraction.

Eventually this process ends as the entire file has transferred to the VMI application and is stored off on the monitor's file system. Note that the previously mentioned checksum is only intended to detect transmission errors, not to provide any means to prove cryptographic integrity as required in applications for digital forensics. Expanding the protocol, in particular, the in-guest agent for this purpose however seems plausible, yet outside the scope of our current work.

6 Evaluation

In this section, the performance of the file extraction mechanism is measured and evaluated. Additionally, the stealthiness of the architecture is elaborated upon in the context of an attacker within the guest system. All tests are performed on virtual machines equipped with one pinned core of an Intel i7-6700K processor and 2048 MB of RAM, swapping is disabled. Both the MVM and PVM system are virtualized by XEN 4.13 using the *Intel VT-x* processor extension. The PVM is located in DomU, while the Dom0 acts as MVM. The PVM uses the Linux kernel version 4.4.40, and the MVM uses version 4.19.0. The system is installed on a Samsung PM951 128 GB SSD, the DomU is stored within a *qcow2* image. The measurements are performed while *CR3*-monitoring is enabled in *libvmi*.

6.1 Transfer Speed

To evaluate the performance of the protocol and its sample implementation, the first thing to measure is the transfer speed when extracting a file from the

guest virtual machine compared to loading the file directly from disk. For this purpose, several files of different sizes are placed in the guest machine as potential extraction targets. We measure the duration of file transmission starting from the request to the agent until the file has been received by the monitor. The following sizes have been selected to represent different classes of files that one may want to extract from the guest: 2 KiB, 1 MiB and 300 MiB.

Then four measurements per file are performed with different buffer sizes:

1. **Native**: Reading the file into a contiguous buffer in the guest without the use of VMI.
2. **One page**: The buffer size is set to one page (4 KiB on the evaluated system) so that the measurement shows the highest possible slowdown due to mutual exclusion.
3. **10 MiB**: The buffer size is set to 10 MiB, a good middle-ground for most applications.
4. **File size**: The buffer size is set to the file size, as this measurement will show potential, inherent slowdowns of the approach that are not caused by mutual exclusion.

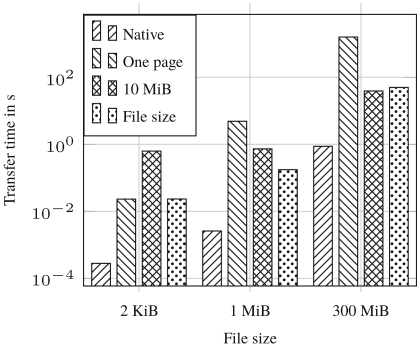

Fig. 5. Transfer time of VMI-aided file extraction Transfer time of VMI-aided file extraction

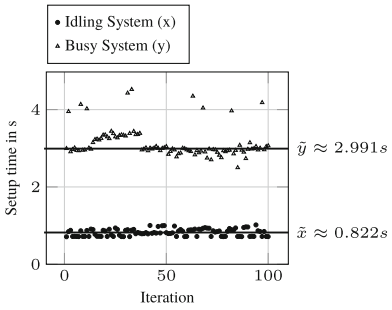

Fig. 6. Scheduler impact on agent deployment

Figure 5 depicts the results of this measurement with a sample size of 10. Given these measurements, we observe a best-case transfer rate of approximately 76 MiB/s with an average of 52 MiB/s. For this estimate, the values of one page buffer size and 2 KiB total file size were not considered. The reason for this decision is that the former is not suitable for general use due to the number of context switches required and only designed to show the worst-case performance of around 1.8 MiB/s. In the latter case however, the run-time is vastly impacted by the setup of the transfer, not by actually exchanging the buffers. The error of these measurements is around 2%, which is not representable in the figure.

While these results can already be considered acceptable, there is still room left for improvement. First, we can disable event handling while the file transfer is in progress. As seen in Sect. 6.3, listening for *CR3* events introduces a huge overhead, which can, therefore, be avoided. Second, the current implementation of the spinlock can be replaced with alternatives that make use of the relevant x86 instructions. Moreover, it seems feasible to use an interrupt for the communication direction PVM to MVM instead.

6.2 Agent Deployment

Additionally, we have to consider the cost of deploying the agent within the production virtual machine in the first place. For this purpose, we measure the duration of ELF injection for our in-guest agent implementation. As this procedure relies on scheduler timing, the results may vary depending on factors such as overall activity in the machine and the specific implementation of the scheduler. We performed these measurements a hundred times on an idling system and a busy system using the default Linux scheduler.

The results of these measurements are depicted in Fig. 6. We find that our assumption of a large fluctuation in the injection time due to scheduler timing is in line with the measured values. The measurements show an expected median setup time of 0.82 s on an idling system, 2.99 s on the busy system. In the worst case, the setup procedure took 1.01 and 4.51 s, respectively. Since the agent can potentially be reused for multiple file transfers, we consider these results to be reasonable.

6.3 Performance Degradation

Furthermore, the execution of the in-guest agent can cause noticeable performance degradation for other applications running in the PVM. To measure potential side effects of the file extraction procedure, a heavy computational load is simulated by executing calculations on the guest. This is done by approximating π with the Chudnovsky algorithm for the first eight iterations [4]. By comparing the computation time under file extraction to normal conditions, any potential slowdowns in the guest system that are not caused by I/O operations become visible. The only way to effectively eliminate the latter problem is to create an artificial bottleneck for the agent's file access. Since this is in direct contrast to the goal of high transfer speeds, I/O throughput is not considered for potential slowdowns. The measurement is repeated 10 times for each instance.

In total, this results in computation time of 2.948 s for normal execution and 3.495 s for file extraction with a respective standard deviation of 0.041 and 0.076 s. Therefore, the use of VMI-based file extraction degrades the guest's performance by approximately 16%. This degradation is mostly caused by the event handling for *CR3* writes. It might be desirable to filter relevant operations in the hypervisor so that the additional context switch for non-monitored processes is no longer required. However, even with this it can be considered unlikely to improve the performance much further as a *VM-exit* is required in all cases to

provide the necessary isolation. What is possible nonetheless, is to disable event handling after the code injection.

6.4 Stealthiness

As previously discussed, the approach remains vulnerable to some detection vectors. First of all, the presence of the hypervisor is detectable from the guest by the use of timing attacks [3]. Because instructions like *cpuid* cause a *VM-exit* on Intel CPUs, the elapsed time will be many times greater when a hypervisor is present. However, the presence of the hypervisor itself is common in many environments and does therefore not provide sufficient reason to suspect a maleficent hypervisor abusing the inherent isolation of this technique to extract files. Nevertheless, some known timing attacks allow the detection of virtual machine introspection as employed by the approach at hand [29].

Unlike other inter-VM communication channels such as *ivshmem* and *ZVIM*, our protocol does not introduce a PCI device, but exchanges data directly through the main memory [1,19]. This design choice enhances the stealthiness of the channel as it requires no direct modification of the VM and prevents the detection of suspicious PCI devices.

Additionally, it seems conceivable to monitor the file system for access to sensitive files. Most operating systems allow event-based notifications for certain actions on the file system, on Linux this is provided by the *inotify* API. Using this method, it is possible to detect the in-guest agent as it attempts to read a monitored file into memory. Since the virtual machine introspection approach grants kernel access, it seems plausible to prevent this possibility of detection by placing a hook on the dispatching function from the VMI application. However, access to these sensitive files could also be visible on file systems that are capable of tracking access time separately from modification time, such as *zfs*.

Furthermore, the code injection technique used to both fork and execute the covert in-guest agent is easily detectable from inside the guest. Because the injection mechanism partially overwrites the *.text* section when performing a context switch to the user-mode program, it appears plausible to check the integrity of the program by comparing the *.text* section to the program on disk when the shellcode performs relevant system calls. This technique can detect the injection mechanism even without enforcing strict code singing. To avoid such attempts, the injected shellcode can be dynamically rewritten using *return-oriented-programming*, leaving the entire *.text* section intact [18].

Finally, the injected in-guest agent itself runs within the guest virtual machine and can, therefore, be detected and potentially attacked. Since the introspection approach allows for kernel access as mentioned above, it is possible to use *DKOM* to hide the forked process and its code from other programs within the guest.

7 Related Work

In the following Section, we evaluate previous work that aims to achieve a similar goal. Table 1 represents an overview over all discussed approaches.

In 2009 Richard Jones introduced a set of tools for accessing and modifying disk images of virtual machines called *libguestfs* [16]. Specifically, the *virt-copy-out* utility enabled file extraction from disk images mounted inside a virtual machine. However, as this tool is not based around introspection, access to encrypted, virtual or network file systems is not possible and applications on live virtual machines are highly limited.

The same year Maartmann et al. demonstrated a technique for extracting cryptographic keys from main memory [17]. One of the use-cases examined for their methods was disk encryption through *TrueCrypt*. By extracting the cryptographic key used in the encryption, the attacker can gain access to sensitive data. Since VMI operations typically operate on the main memory, this approach can easily be adapted for use in VMI environments.

Gu et al. showcased an active introspection framework for narrowing the semantic gap by executing ELF binaries in the context of a production virtual machine in 2011 [9]. This was realized by using the ELF loader on the monitoring virtual machine to load a statically linked program to the main memory. By hooking into the scheduler of the production virtual machine using CR3 monitoring, they implemented context switching on-top of the production operating system. We show that by using the program loader and scheduler inside the PVM instead, we can significantly increase performance and reduce the requirements on the implanted program while decreasing isolation.

Soon after, Hale et al. released the *GEARS* framework for VMI-based services. They argued that such services should place components inside the non-compliant production virtual machine. By doing so, the implementation can be simplified as programs running inside the production virtual machine do not suffer from the semantic gap. This principle is fulfilled in our work through the use of the dynamically injected in-guest agent.

Fu et al. proposed a compatibility layer for non-VMI applications called *HyperShell* in the same year [6]. They introduced the concept of *reverse system calls* that allowed them to selectively forward some system calls to the production virtual machine while executing others on the monitoring virtual machine. This compatibility layer essentially enables the reuse of existing binaries such as *cp*, *ls*, etc. in VMI contexts, thus greatly simplifying VM management. However, the architecture shows weaknesses in terms of compatibility and portability: First, the concept of *reverse system calls* inevitably requires the same (or at least a compatible) set of system calls. This means *HyperShell* cannot be used in situations where the MVM and PVM run different operating systems. Furthermore, the implementation requires modifications to the hypervisor, which presents an obstacle in practical real-world applications where the hypervisor cannot be patched for security and liability reasons.

Morbitzer et al. introduced a technique based on their previously published *SEVered* attack to extract encryption keys for file systems and other applications

Table 1. Comparison with related work

	Arbitrary code execution[4]	ELF injection	Trusted Execution Environment	Hard disk file systems	Encrypted file systems	Other file systems[5]	wo/ VMM modifications	wo/ compatible system calls
libguestfs [16]	✗	✗	✗	✓	✗	✗	✓	✓
Maartmann et al. [17]	✗	✗	✗	✓	✓	✗	✓	✓
Morbitzer et al. [21]	✗	✗	✓	✗	✓	✗	✓	✓
Gears [10]	✓	✗	✗	✗	✗	✗	✗	✓
Process Implanting [9]	✓	✓	✗	✗	✗	✗	✗	✓
HyperShell [6]	✓	✗	✗	✓	✓	✓	✗	✗
Agent-based extraction	✓	✓	✗	✓	✓	✓	✓	✓

[4] Arbitrary code execution means that the application can dynamically inject code into the PVM. However, this does not necessarily indicate that binary formats like ELF can be executed.

[5] For this comparison, we refer to network file systems, virtual file systems and temporary file systems as other file systems.

using virtual machine introspection in 2019 [21,22]. Their approach enabled the extraction of sensitive data when the virtual machine was protected by *AMD SEV* that encrypts the main memory of the VM with a key unknown to the hypervisor. This enables file extraction, even in areas not covered by our file extraction architecture. However, the presented approach falls short when dealing with file systems that reside purely in RAM such as *tmpfs* or are simply not accessible by the monitor such as *WebDAV*.

8 Conclusion

This paper addresses the adaptation of typical code injection techniques for VMI-based applications and the extraction of files from virtual machines through the use of an introspection-oriented in-guest agent. To address the issue of deploying the in-guest agent in the targeted virtual machine, we show how typically used techniques for inter-process code injection can be adapted for inter-machine applications using introspection. Furthermore, the implementation for Linux MVM/PVM-systems is presented.

For obtaining files that are accessible from within the virtual machine, our approach demonstrates the provisioning and placement of an in-guest agent within the guest. This in-guest agent enables common memory forensic techniques and tools to access non-volatile storage. Additionally, the presented solution is evaluated in terms of transfer speed, performance degradation and stealthiness.

Acknowledgments. This work has been funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 361891819 (ARADIA).

References

1. Armbruster, M.: Device Specification for Inter-VM shared memory device (2016). https://fossies.org/linux/qemu/docs/specs/ivshmem-spec.txt. Accessed 27 July 2020
2. Bahram, S., et al.: DKSM: subverting Virtual Machine Introspection for Fun and Profit. In: 29th IEEE Symposium on Reliable Distributed Systems, pp. 82–91. IEEE (2010)
3. Brengel, M., Backes, M., Rossow, C.: Detecting hardware-assisted virtualization. In: Caballero, J., Zurutuza, U., Rodríguez, R.J. (eds.) DIMVA 2016. LNCS, vol. 9721, pp. 207–227. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40667-1_11
4. Chudnovsky, D., Chudnovsky, G.: Approximations and complex multiplication according to Ramanujan. In: Pi: A Source Book, pp. 596–622. Springer, New York (2004). https://doi.org/10.1007/978-1-4757-4217-6_63
5. Dolan-Gavitt, B., Leek, T., Zhivich, M., Giffin, J., Lee, W.: Virtuoso: narrowing the semantic gap in virtual machine introspection. In: IEEE Symposium on Security and Privacy, pp. 297–312. IEEE (2011)
6. Fu, Y., Zeng, J., Lin, Z.: HYPERSHELL: a practical hypervisor layer guest OS shell for automated in-VM management. In: Proceedings of the 2014 USENIX Annual Technical Conference, pp. 85–96. USENIX Association, USA (2014)
7. Garfinkel, T., Rosenblum, M.: A virtual machine introspection based architecture for intrusion detection. In: Proceedings of the Network and Distributed System Security Symposium. The Internet Society (2003)
8. Gorman, M.: Understanding the Linux Virtual Memory Manager, vol. 352, 1st edn., pp. 33–38. Prentice Hall, Prentice (2004)
9. Gu, Z., Deng, Z., Xu, D., Jiang, X.: Process implanting: a new active introspection framework for virtualization. In: 2011 IEEE 30th International Symposium on Reliable Distributed Systems, pp. 147–156. IEEE (2011)
10. Hale, K.C., Xia, L., Dinda, P.A.: Shifting GEARS to enable guest-context virtual services. In: Proceedings of the 9th International Conference on Autonomic Computing, New York, NY, USA, pp. 23–32. Association for Computing Machinery (2012)
11. Hebbal, Y., Laniepce, S., Menaud, J.: Virtual machine introspection: techniques and applications. In: 10th International Conference on Availability, Reliability and Security, pp. 676–685. IEEE (2015)
12. Herrmann, D.: memfd_create - create an anonymous file. In Linux Programmer's Manual (2020). http://man7.org/linux/man-pages/man2/memfd_create.2.html. Accessed 09 Feb 2020
13. Hussein, N.: Randomizing structure layout (2017). https://lwn.net/Articles/722293/ (2017). Accessed 01 Jan 2020
14. Intel Corporation: Intel® 64 and IA-32 Architectures Software Developer's Manual, vol. 3D, pp. C1–C3, September 2016
15. Jain, B., Baig, M.B., Zhang, D., Porter, D.E., Sion, R.: SoK: introspections on trust and the semantic gap. In: IEEE Symposium on Security and Privacy, pp. 605–620. IEEE (2014)

16. Jones, R.: libguestfs - tools for accessing and modifying virtual machine disk images (2009). http://libguestfs.org/. Accessed 27 July 2020
17. Maartmann-Moe, C., Thorkildsen, S.E., Årnes, A.: The persistence of memory: forensic identification and extraction of cryptographic keys. In: Digital Investigation, vol. 6, pp. S132–S140. Elsevier (2009)
18. McNamara, R.: Linux based inter-process code injection without ptrace(2) (2017). https://blog.gdssecurity.com/labs/2017/9/5/linux-based-inter-process-code-injection-without-ptrace2.html. Accessed 01 Jan 2020
19. Mohebbi, H., Kashefi, O., Sharifi, M.: ZIVM: a zero-copy inter-VM communication mechanism for cloud computing. In: Computer and Information Science, vol. 4, pp. 18–27. Canadian Center of Science and Education (2011)
20. Molnár, I.: This is the CFS scheduler (2007). http://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt. Accessed 19 Jan 2020
21. Morbitzer, M., Huber, M., Horsch, J.: Extracting secrets from encrypted virtual machines. In: Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy, CODASPY 2019, pp. 221–230. Association for Computing Machinery (2019)
22. Morbitzer, M., Huber, M., Horsch, J., Wessel, S.: SEVered: subverting AMD's virtual machine encryption. In: Proceedings of the 11th European Workshop on Systems Security, EuroSec@EuroSys, pp. 1–6. Association for Computing Machinery (2018)
23. Payne, B.D.: Simplifying virtual machine introspection using libvmi. Technical report, Sandia National Laboratories (2012)
24. Pfoh, J., Schneider, C.A., Eckert, C.: A formal model for virtual machine introspection. In: VMSec 2009. Association for Computing Machinery (2009)
25. Rekall: Rekall memory forensics framework (2012). https://github.com/google/rekall (2012). Accessed 07 Oct 2019
26. Tanenbaum, A.S.: Modern Operating Systems, pp. 74–75. Prentice Hall, Prentice (2004)
27. Taubmann, B., Rakotondravony, N., Reiser, H.P.: Cloudphylactor: harnessing mandatory access control for virtual machine introspection in cloud data centers. In: IEEE Trustcom/BigDataSE/ISPA, pp. 957–964. IEEE (2016)
28. Taubmann, B., Rakotondravony, N., Reiser, H.P.: Libvmtrace: Tracing virtual machines. Winter School on Operating Systems (WSOS) (2016). WiP abstract
29. Tuzel, T., Bridgman, M., Zepf, J., Lengyel, T.K., Temkin, K.J.: Who watches the watcher? Detecting hypervisor introspection from unprivileged guests. In: Digital Investigation, vol. 26, pp. S98–S106. Elsevier (2018)
30. Volatility Foundation: Volatility memory forensics framework (2009). https://github.com/volatilityfoundation/volatility. Accessed 07 Oct 2019