Leon *et al. Cybersecurity* (2021) 4:19
https://doi.org/10.1186/s42400-021-00083-9

Cybersecurity

RESEARCH Open Access

Hypervisor-assisted dynamic malware analysis

Roee S. Leon[1], Michael Kiperberg[2], Anat Anatey Leon Zabag[2] and Nezer Jacob Zaidenberg[3,4*]

Abstract

Malware analysis is a task of utmost importance in cyber-security. Two approaches exist for malware analysis: static and dynamic. Modern malware uses an abundance of techniques to evade both dynamic and static analysis tools. Current dynamic analysis solutions either make modifications to the running malware or use a higher privilege component that does the actual analysis. The former can be easily detected by sophisticated malware while the latter often induces a significant performance overhead. We propose a method that performs malware analysis within the context of the OS itself. Furthermore, the analysis component is camouflaged by a hypervisor, which makes it completely transparent to the running OS and its applications. The evaluation of the system's efficiency suggests that the induced performance overhead is negligible.

Introduction

Malicious software, or malware, refers to a program that is intended to cause damage to the host computer. The malware may attack the computer on which it is executed as well as the computers to which this computer is connected (e.g., via a computer network). Many types of malware exist, including viruses, spyware, adware, rootkits, trojans, ransomware (Hull et al. 2019), and so on (Vinod et al. 2009).

Two basic approaches exist for malware analysis: The first involves static analysis techniques, while the second involves dynamic analysis techniques (Gandotra et al. 2014; Saeed et al. 2013). A common static analysis technique is based on signatures and is typically used by virus scanners. These tools use a database of known potentially malicious instruction sequence patterns. A program is considered malicious if it matches one or more of these patterns. However, the malware uses an abundance of evasion techniques to avoid detection by this type of tool. The common evasion technique is obfuscation, in which the malware attempts to camouflage itself (O'Kane et al. 2011). An obfuscation method commonly used by malware is runtime polymorphism (O'Kane et al. 2011), in which the malware changes its appearance, in run-time, to avoid detection. An example of polymorphic malware is malware that encrypts or packs its malicious code and only decrypts or unpacks it during execution. The decryption code itself is obfuscated using code transformation techniques such as control flow flattening and instruction substitution (You and Yim 2010). Another obfuscation method commonly used by malware is metamorphism (You and Yim 2010; Rad et al. 2012; Basya et al. 2013), in which the malware recodes itself every time it needs to propagate. To achieve this, metamorphic malware may also use code transformation techniques but in a granularity that is typically larger than polymorphic malware (e.g., by transforming the entire executable code).

The most prominent problem of the signature-based tools is the fact that they ignore the semantics of the program. To address this issue, a class of *semantic-aware* malware detectors was introduced (Christodorescu et al. 2005; Feng et al. 2014). These tools check potential malware against an abstract model (Kinder et al. 2005) to determine whether it has malicious intentions, thus achieving higher detection rates of polymorphic or metamorphic malware. Recently, it was shown that even the most sophisticated static analysis tools can be evaded by

*Correspondence: scipio@scipio.org
[3]College of Management Academic Studies, Rishon LeTsiyon Israel
[4]University of Jyväskylä, Jyväskylä, Finland
Full list of author information is available at the end of the article

malware (Moser et al. 2007). The authors of (Moser et al. 2007) present obfuscation transformations, using special primitives they refer to as *opaque constants*, with which malware can evade even the most sophisticated static analysis tools.

To overcome the limitations of the static analysis tools, many systems use a dynamic approach for malware analysis. These systems are often referred to as *behavior-based*, since they analyze the run-time behavior of the malware (Egele et al. 2008). Typically, user-space malware uses OS services intensively to perform some meaningful work (e.g., sending a file over the network). Therefore, systems that use a dynamic approach must intercept these calls to build a meaningful behavioral profile. The criteria on which these systems are built are: (1) efficiency, (2) transparency, and (3) quality of the analysis. Efficiency is important, as the behavioral profile of malware may need to be extracted at a reasonable time. For example, some malware intentionally slows themselves down to increase the detection time significantly. This is especially true in slow environments, such as in an emulator. Transparency is also important, as sophisticated malware may attempt to detect sandbox environments. (Yokoyama et al. 2016). The quality of the analysis primarily depends on the extracted information and on the tool that analyzes this information.

Current dynamic malware analysis methods can be divided into four classes: (1) hooking methods, (2) emulation methods, (3) hypervisor-based methods, and (4) bare-metal based methods. Hooking methods (Willems et al. 2007; Guarnieri and Fernandes 2010; Yalew et al. 2017) perform inline overwriting of API code directly in the process memory. Therefore, the malware attempts to use any of the Windows APIs can be monitored. This is typically done by injecting a special module into the monitored process address space. Though these methods are very efficient, they also have several significant deficiencies. These methods are:

1. detectable — since these methods directly modify the process memory, malware can simply check the contents of the desired API function against a known signature to check whether it was changed. For example, the structure of the Native API (i.e., *ntdll.dll*) stubs can be easily determined and have a similar structure in different Windows versions. Also, the monitoring module may be detected by asking the OS for a list of loaded modules (see *skippable*).
2. modifiable — malware can simply remove or replace the hooking code by writing directly into the process memory. If *DEP* (Data Execution Prevention) is in use, the malware may call *VirtualProtect* to get write permissions (see *skippable*).
3. skippable — malware can skip Windows API calls by performing system calls directly. Even though the system call numbers vary between different version of Windows, we claim that this is completely feasible, as it is sufficient to use the *NtQueryValueKey* system call to detect the exact Windows version.

Emulation methods, e.g., (Bayer et al. 2005), perform system call tracing without inducing any modifications to the emulated environment. However, these methods are slow and can be exploited due to incomplete emulation (Dinaburg et al. 2012; Ferrie 2006; Raffetseder et al. 2007; Vidas and Christin 2014).

Many modern malware analysis tools (Zaidenberg 2018) use hypervisor-based methods. Tools such as (Dinaburg et al. 2012; Lengyel et al. 2014) provide a malware analysis system that is both transparent (i.e., perform no modifications to the running OS) and are more efficient compared to emulation methods. Both (Dinaburg et al. 2012) and (Lengyel et al. 2014) are built on top of the XEN hypervisor (Barham et al. 2003) and consequently run the malware sample inside a guest domain while the actual analysis takes place in the control domain (dom0). In both methods, every conducted system-call requires multiple transitions to the hypervisor. As a consequence, these methods incur significant overhead and are also vulnerable to attacks on the XEN hypervisor (Ding et al. 2012). Moreover, the bare existence of the XEN hypervisor also incurs a non-negligible overhead (Li et al. 2013). All hypervisor-based methods are vulnerable to VMM detection attacks (Garfinkel et al. 2007; Branco et al. 2012; Franklin et al. 2008).

Bare-metal based methods (Kirat et al. 2011; Kirat et al. 2014) provide a bare-metal system for malware monitoring and are therefore not prone to hypervisor detection attacks. However, extraction of a behavioral profile of malware from such systems is difficult, because an analysis component must be installed on the target machine. The presence of such a component can be detected by sophisticated malware. Therefore, these methods are best suited for tracking activities that can be externally monitored (e.g., network and disk activities).

Our method can be classified as a hypervisor-based method. However, in our method:

1. The system call monitoring component is located within the guest OS and consequently no intervention of the hypervisor is required during the system-call monitoring and analysis. As a result, the performance overhead is kept to a minimum.
2. Though completely handled in the guest context, the monitoring component is fully transparent to the guest OS.

The code of the hypervisor and the monitoring component consists of 8,000 lines of code. This code size is significantly smaller than that of other hypervisors (Murray et al. 2008; VMWare 2005). Moreover, our hypervisor provides no direct API calls (e.g., through VM-call) to the guest OS. We believe that the latter points improve the reliability of our hypervisor.

To enable these claims, our method uses a hypervisor. A hypervisor is a software module that can monitor and control the execution of an OS. These capabilities are provided by an extension to the original processor's instruction set, called "virtualization extensions," which are available on processors designed by Intel (VT-x) (Intel Corporation 2007), AMD (AMD-V) (AMD 2010), and ARM (Virtualization Extensions) (ARM Ltd. 2013) and other architectures.

In the past we have shown how hypervisors can be used to obtain memory contents (Ben Yehuda et al. 2021) for forensics analysis and also create honeypots (Zaidenberg et al. 2020) that encourage malware to expose themselves.

Interception of system calls by a hypervisor may induce a significant overhead (Li et al. 2013; Dinaburg et al. 2012). Our method uses a novel approach, in which a monitoring component is injected into the guest OS memory. The monitoring component is responsible for the malware monitoring and analysis. The hypervisor protects the monitoring component from modification and detection using a technique similar to *invisible-breakpoints* described in (Deng et al. 2013). Our method is implemented on Intel processors but can be easily ported to AMD and ARM. The evaluation of our method suggests that the performance overhead of the system is negligible.

Garfinkel et al. (2007) illustrate hypervisor detection possibilities using logical, resource, and timing discrepancies. Attacks that are based on these discrepancies are outside the scope of this paper.

Throughout this paper, we refer to the entity that wants to monitor the behavior of a given process as the *malware analyst*.

Our main contributions can be summarized as follows:

We propose a method that has the advantages of an in-guest malware analysis tool (e.g., CWSandbox (Willems et al. 2007)) yet overcomes its disadvantages using hardware virtualization. Given that the running kernel is trusted (Zaidenberg et al. 2015)), the guest component can use various OS functionality to bridge the semantic gap. Due to the latter, it is also trivial to store the system-call trace by any means supported by the OS (e.g., USB, Network, Disk, etc.). In contrast, other solutions might require to perform full virtualization to these hardware-devices thus resulting in even greater performance overhead.

Last, in the digital forensics world it is not uncommon for separate process to collect data (e.g. software such as LiME for memory analysis (Sylve 2012), rekall (Cohen 2014) Snort sniffer and packer logger (Roesch 1999) etc. and a separate process to perform the analysis of memory (e.g. volatility (Case and Richard III 2017) or network data (White et al. 2013), (Shah and Issac 2018)) Such software can be based on state machine or machine learning methods (Kharaz et al. 2016), (Ucci et al. 2019). The software described in this paper fills the role of the data collector. This tool can also be used to detect some attacks but it is recommended that a secondary tool, possibly AI based will be used to detect anomalies.

VMX

Many modern processors are equipped with a set of extensions to their basic instruction set architecture that enables them to execute multiple OSs simultaneously. This paper discusses Intel's implementation of these extensions, which they call Virtual Machine Extensions (VMX). The software that governs the execution of these operating systems is called a hypervisor, and each OS (with the processes it executes) is called a guest. Transitions from the hypervisor to the guest are called VM-entries and transitions from the guest to the hypervisor are called VM-exits. While VM-entries occur voluntarily at the instigation of the hypervisor, VM-exits are caused by some event that occurs during the guest's execution. The events may be synchronous, for example, execution of an INVLPG instruction, or asynchronous, such as a page-fault or general-protection exception. The event that causes a VM-exit is recorded for future use by the hypervisor. A special data structure called the Virtual Machine Control Structure (VMCS) allows the hypervisor to specify the events that should trigger a VM-exit as well as many other settings of the guest.

Intel's Extended Page Table (EPT), a technology generally called Secondary Level Address Translation (SLAT) allows the hypervisor to configure the mapping between the physical address space as it is perceived by a guest and the real physical address space. Similarly to the virtual page table, EPT allows the hypervisor to specify the access rights for each guest's physical page. When a guest attempts to access a page that is either not mapped or has inappropriate access rights, an event called EPT-violation occurs, triggering a VM-exit.

System description

The system described in this paper consists of four components: (1) a boot application, (2) a hypervisor, (3) a monitoring component, and (4) a process behavior analyzer. The boot application is responsible for initializing a hypervisor. The boot application is implemented as a UEFI or MBR application (depending on the firmware). The hypervisor is responsible for installing the monitoring component and protecting it from detection and modification. The monitoring component, in turn, is

responsible for system call intercepting and recording. A special sandbox configuration file determines the system calls recording format. Finally, the process behavior analyzer is responsible for analyzing the output of the monitoring component.

Preparations

The malware analyst must set up a malware analysis lab, that is, a lab that allows full control over the malware potential operation (e.g., network access). Next, the malware analyst must create a sandbox configuration file and, finally, install the necessary files on the target machine. Setting up a malware analysis lab is beyond the scope of this paper. The other two processes are described in the following subsections.

Sandbox configuration file

A system call typically receives a list of parameters that describe the request of the caller. For example, *NtOpenFile* receives six arguments, of which four are input arguments that describe the properties of the file to be opened (e.g., its path, desired access, etc.). The length of the argument list is limited. This limit is different for each OS. For example, in Windows 7 x86, the maximum number of arguments a system call may receive is 17. A trivial system call analyzer may use this fact to simply copy the first 17 arguments of each system call. A special analyzer may later be used to filter the unnecessary arguments of each system call. However, this method of system call analysis is ineffective as a system call parameter may potentially point to a hierarchy of meaningful information. For example, the third parameter of *NtOpenFile* is a pointer to a *POBJECT_ATTRIBUTES* structure, which, in turn, has a field that points to a *PUNICODE_STRING* structure, which, in turn, has a field that points to a buffer that contains the actual path of the file to be opened; see Fig. 1. To describe more complex relations, the malware analyst must create a formatted configuration file that describes these relations.

The sandbox configuration file allows the description of the complex relations between parameters and supports unlimited depth. It supports four-parameter types: (1) primitive, (2) pointer, (3) structure, and (4) buffer (pointer and length). To ease the analysis process, the sandbox configuration file supports the naming of structure fields and system call parameters (see "Evaluation" section). For simplicity, the sandbox configuration file must be created manually by the malware analyst. The creation process can be almost fully automated; this is described in "Limitations and future work" section. Figure 2 presents a simple sandbox configuration file (valid for Windows 7 x86) containing a single system call entry. Line 1 provides information regarding the specific system call. Specifically, its number and the number of relevant parameters. Lines 2, 3, 8, and 9 provide information regarding these parameters. Each parameter line is composed of

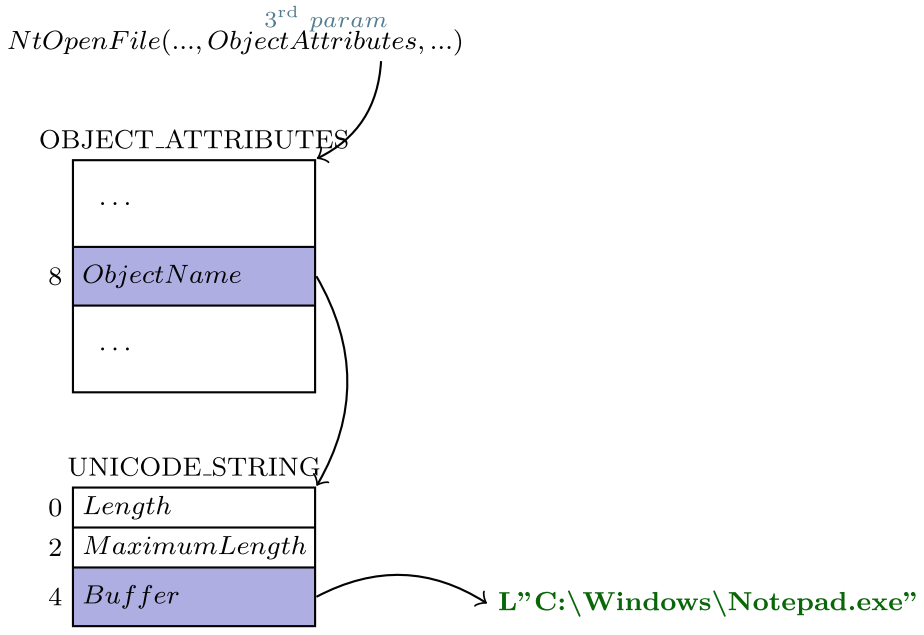

Fig. 1 NtOpenFile third parameter information hierarchy in a x86 Windows system. The *ObjectAttributes* parameter points to an *OBJECT_ATTRIBUTES* structure. A field named *ObjectName* is located at an offset of 8 bytes in the *OBJECT_ATTRIBUTES* structure. This field points to a *UNICODE_STRING* structure. A field named *Buffer* is located at an offset of 4 bytes in the *UNICODE_STRING* structure. This field points to the unicode string "C:\Windows\Notepad.exe"

```
1.  s,179,4 ; NtOpenFile
2.  p,1,pri,4,ep ; (ACCESS_MASK) DesiredAccess
3.  p,2,ptr,p ; (POBJECT_ATTRIBUTES) ObjectAttributes
4.          p,struct,f ; (OBJECT_ATTRIBUTES)
5.                  f,8,ptr,p ; (PUNICODE_STRING) ObjectName
6.                  p,struct,f ; (UNICODE_STRING)
7.                          f,4,buffer,0,2,ef ; (PWSTR) Buffer
8.  p,4,pri,4,ep ; (ULONG) ShareAccess
9.  p,5,pri,4,ep ; (ULONG) OpenOptions
```

Fig. 2 Sandbox configuration file example for Windows 7 x86

the parameter index (0 for the first parameter, 1 for the second, and so on), its type, and type-specific data. For example, line 2 describes the second parameter of *NtOpenFile*, *DesiredAccess*. The parameter is a primitive type, with a size of 4 bytes. Lines 4 and 6 describe a pointee of a structure type. Each structure type is composed of one or more fields. For example, line 4 describes a structure of type *OBJECT_ATTRIBUTES*, which is pointed by the third parameter (*ObjectAttributes*). The structure is composed of a single pointer field at an offset of 8 bytes. This field points to another structure field of type *UNICODE_STRING*, which is composed of a single field. This field is located at an offset of 4 bytes and is of the buffer type. Text that appears after a semicolon on each line describes the field or parameter name and type. This text is ignored by the parser and is only used by the process behavior analyzer.

Target installation

The boot application is implemented as a UEFI or MBR application (depending on the firmware).

In UEFI, after a successful startup, the UEFI boot manager loads a sequence of executable images, called UEFI applications. The UEFI firmware stores the location in which these images reside in non-volatile storage. The boot-sequence can be configured using the firmware setup screen. The UEFI boot manager loads an executable image into the main memory, undertakes the necessary fixups, and executes its main routine. In case the entry routine returns, the UEFI boot manager proceeds to the next executable image if there is one. The UEFI application's entry routine may also not return. A typical example of the latter is an OS loader implemented as a UEFI application.

In MBR, the firmware attempts to locate a bootable device according to the defined boot order, which is typically stored on a CMOS chip. A bootable device is identified by a special sector at its very beginning. This sector is referred to as the Master Boot Record (MBR). The MBR contains up to four partitions, of which one must be marked as active (typically, the OS bootloader). Once an active partition is found, the BIOS loads its first sector to a pre-defined location and begins to execute it.

The system described in this paper can be implemented either as a UEFI or as an MBR application. For example, one may choose an MBR application due to a lack of support from the OS (e.g., Windows 7 x86 does not support UEFI boot). A malware analyst interested in installing the system must install the boot application and the configuration file into a location accessible by the firmware. For example, in UEFI, this location lies in the ESP partition on which the application resides.

Operation

The boot application, during the boot phase, obtains the configuration file from the disk, initializes a hypervisor, and returns to the original OS bootloader. The hypervisor remains in the main memory and continues its operation even after the application terminates. The hypervisor maps a monitoring component into the address space of the guest OS, and the latter intercepts all system calls and records only those that belong to the monitored process. The hypervisor then protects the monitoring component from detection and modification. The remainder of this section provides a detailed explanation of the initialization and operation of the system.

HV initialization

The boot application starts by allocating a persistent memory block. In UEFI, this is done by the allocation functions it provides. In BIOS, this is done by modifying the BIOS memory map; simply put, the application finds a memory region that is large enough to hold the hypervisor and subtracts the size of the hypervisor from the region's length. During the hypervisor initialization, the *EPT* and the *IOMMU* are set up. The hypervisor sets the *EPT* and the *IOMMU* mappings via the following steps:

1. The hypervisor sets an identity mapping between the real and the guest physical address spaces. The latter

is done by configuring the EPT such that the guest physical page X translates to host physical page X. Fortunately, setting up identity mapping between the real physical address space and the I/O peripherals physical address space is trivial, as the page-table hierarchy used by the EPT can also be used by the IOMMU.

2. The hypervisor sets the access rights of its code and data to read-only. This step ensures that malicious code, even if it executes in kernel mode, cannot modify the hypervisor's code and data. Figure 2 depicts the physical address space as it is perceived by the guest and I/O peripherals after HV initialization.

Monitoring component initialization

Efficient interception of all system-calls by a hypervisor is not a simple task. In x86 and x86-64, a special Model Specific Register (MSR) is used to indicate the virtual address of the system call handler; these MSRs are referred to as *SYSENTER_EIP* and *LSTAR*, respectively. To intercept system calls, a hypervisor could intercept each instance of access to these MSRs. However, this method is inefficient as it implies a VM-exit and a VM-entry on each system call.

The performance of system call interception can be improved by performing the interception in the context of the guest OS. Obviously, for the sake of isolation, the address space of the hypervisor is separated from that of the guest OS. For an efficient system call interception, the hypervisor must map the monitoring component into the address space of the guest OS.

The monitoring component does not depend on external libraries. Nevertheless, it can use functions provided by the kernel itself (for example, the *Zw* family in Windows).

A possible approach for mapping the monitoring component into the address space of the guest OS is by directly modifying the OS page tables (specifically, the upper kernel part). However, this method is intrusive and may cause system instabilities (for example, in Windows 10, we observed that the Memory Management Unit of Windows generates a BSOD on such attempts). An alternative method is to virtualize the page-tables of the guest OS (a technique known as shadow-page-tables). However, this method is both complex and much less efficient.

Because of these deficiencies, we decided to map the monitoring component into the address space of the guest OS as follows. The hypervisor waits for the OS kernel to be successfully loaded. During its initialization, the OS kernel writes to the system call handler register. The hypervisor uses this event as an indication of the OS load and configures the guest such that writes to the system call handler register cause a VM-exit. When such an attempt occurs, the hypervisor looks for the kernel image within the guest memory and uses its export table to find the address of *ExAllocatePool*. Figure 3 depicts the guest memory layout upon a VM-exit caused by writing to the system call handler register. Then, the hypervisor configures the guest to cause a VM-exit upon a breakpoint exception. Next, the hypervisor replaces the first byte of the *ExAllocatePool* function with a breakpoint instruction (`INT3` in x86) and configures the guest such that a VM-exit is triggered upon a breakpoint exception. Algorithm 1 depicts the actions of the hypervisor upon the next call to *ExAllocatePool*. Simply put, the hypervisor replaces the original call arguments with its own arguments (e.g., the amount of memory required for the monitoring component) and saves the original arguments. Upon completion, the hypervisor restores the original arguments and continues the execution of the guest. Finally, when the allocation is complete, the hypervisor forfeits control over the system call handler register.

Finally, the hypervisor hooks the original system call handler such that it calls the monitoring component (which is now mapped into the address space of the guest OS). The hypervisor protects the monitoring component from both modification and detection, as described in "Security and transparency" section.

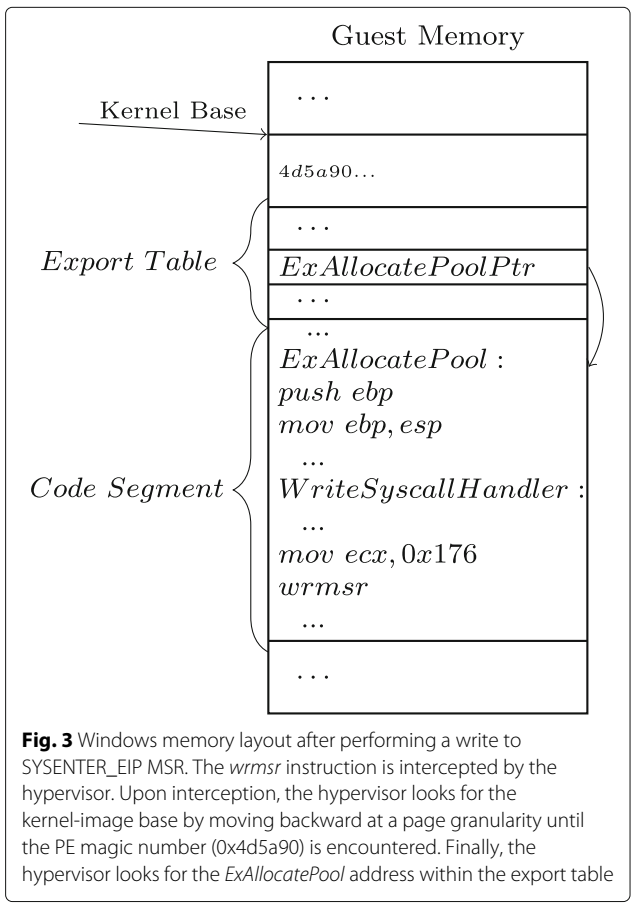

Fig. 3 Windows memory layout after performing a write to SYSENTER_EIP MSR. The *wrmsr* instruction is intercepted by the hypervisor. Upon interception, the hypervisor looks for the kernel-image base by moving backward at a page granularity until the PE magic number (0x4d5a90) is encountered. Finally, the hypervisor looks for the *ExAllocatePool* address within the export table

Algorithm 1 Hypervisor Breakpoint Handler

```
1:  procedure HandleBreakPoint()
2:      if IP is in ExAllocatePool entry then
3:          Save return address
4:          Save original args
5:          Modify original args
6:          Restore original entry byte
7:          Save original return byte
8:          Set return byte to 0xCC
9:      else if IP is in return address then
10:         CopyGuestApplicationToMemory()
11:         Restore original args
12:         Restore original exit byte
13:         Set ip to entry of ExAllocatePool
```

Monitoring component operation

The monitoring component intercepts all system calls and records only those belonging to the monitored process. Initially, the monitoring component sets up a sandbox by parsing the configuration file. A system call conducted by a monitored process is analyzed and then recorded. The recorded data is written into an internal memory buffer and then written to an external source every time the buffer gets full so it can be reused. The external source can be a remote server, a local file, or a large, previously allocated memory buffer. For simplicity, we chose the option of a local file. At the end of each interception, our handler jumps to the original OS handler. Algorithm 2 depicts the monitoring component system call handler. The data for analysis is fetched from the arguments according to the configuration file, provided by the malware analyst, and is sequentially written into a buffer of fixed length. If a configuration is not provided for a system call, the maximum possible number of arguments (e.g., 17 in Windows 7 x86) is fetched. When the buffer becomes full, the data is written to an external source so it can be reused.

Algorithm 2 Monitoring Component — System Call Handler

```
1:  procedure HandleSystemCall(scNo)
2:      args ← GetArgs()
3:      cp ← GetCurrentProcess()
4:      mp ← GetMonitoredProcess()
5:      if NotActive(mp) and not Exit(scNo) then
6:          if IsMonitoredProcess(cp) then
7:              MonitorProcessInit(mp)
8:      else if Active(mp) and Exit(scNo) then
9:          MonitoredProcessDumpAndFree(mp)
10:     if Active(mp) and IsMonitored(cp) then
11:         MonitoredProcessAnalyze(mp, scNo, args)
12:     JumpToOriginalHandler()
```

Security and transparency

Our method can be classified as a hypervisor-based method. However, in our method:

1. The system call monitoring component is located within the guest OS and consequently, no intervention of the hypervisor is required during the system-call monitoring and analysis. As a result, the performance overhead is kept to a minimum.
2. Though completely handled in the guest context, the monitoring component is fully transparent to the guest OS.

The code of the hypervisor and the monitoring component consists of 8,000 lines of code. This code size is significantly smaller than that of other hypervisors (Murray et al. 2008; VMWare 2005). Moreover, our hypervisor provides no direct API calls (e.g., through VM-call) to the guest OS. We believe that the latter points improve the reliability of our hypervisor.

Garfinkel et al. (2007) illustrate VMM detection possibilities using logical, resource, and timing discrepancies. Attacks that are based on these discrepancies are outside the scope of this paper.

The hypervisor protects itself and the monitoring component from subversion. In addition, the hypervisor must assure that the monitoring component remains undetected. These processes are described in the following subsections.

HV memory protection

Secondary Level Address Translation (SLAT) is a mechanism implemented as part of hardware-assisted virtualization technology to reduce the overhead of managing the hypervisor's guest page-tables. Second Level Address Translation is supported by Intel (EPT), AMD (RVI), and ARM (Stage-2 page-tables). Simply put, SLAT allows the hypervisor to control the mapping of physical-page addresses as they are perceived by the guest (known as guest-physical-address) to real physical-pages addresses (known as host-physical-address). An analogy to SLAT usage in a virtualized environment (i.e., controlled by a hypervisor) is virtual page-table usage in a process context in a non- virtualized environment (i.e., controlled by an OS). Figure 3 depicts the guest's address translation process.

The Input-Output Memory Management Unit (IOMMU) is a memory management unit that stands between DMA-capable peripherals and the main memory. In this sense, it functions as a virtual page-table for devices. DMA is a hardware mechanism that allows peripherals to access the main memory directly without going through the processor. The IOMMU allows the OS or hypervisor to set paging structures for the peripherals:

that is, the peripherals will access a virtual address (also known as I/O address) that will be translated by the IOMMU. To protect itself against malicious modifications, the hypervisor configures both the EPT and the IOMMU in such a way that all of its sensitive memory and regions are not mapped and are therefore not accessible from either the guest or from a hardware device.

The hypervisor also needs to protect the monitoring component from modifications. The physical pages in which it is stored cannot be unmapped, as they need to be accessible from the guest OS. Therefore, the hypervisor simply configures the EPT and the IOMMU such that the memory on which the monitoring component resides has only read and execute permissions (Figs. 4 and 5).

Monitoring component transparency

As described in "HV memory protection" section, the SLAT (EPT) mechanism allows the hypervisor to also configure the access rights of the guest's physical memory. Using EPT, the hypervisor can make code modifications that are done to the OS and its applications completely transparent to the guest.

The page on which the OS system call handler resides has been modified by the hypervisor (it contains a hook to the monitoring component). The hypervisor configures that page to have only execute permissions. A guest read or write attempt results in an EPT violation which

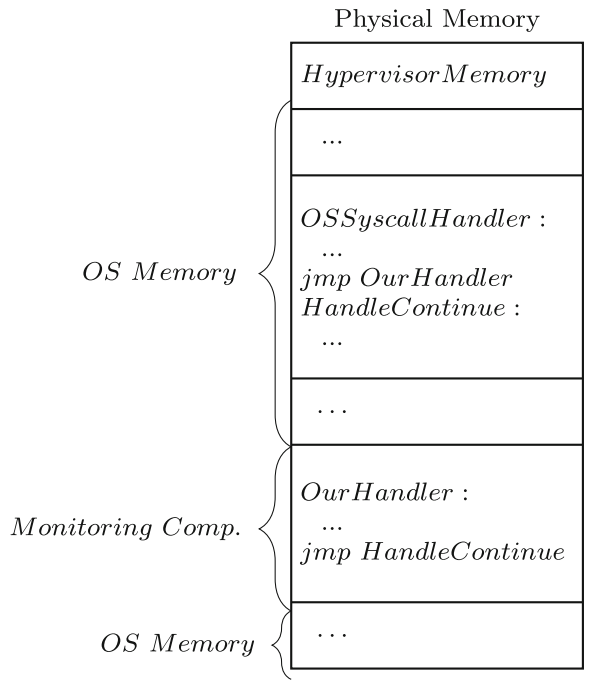

Fig. 4 Physical memory layout after full initialization of the monitoring component. The original OS handler (*OSSyscallHandler*) jumps to our handler (*OurHandler*) during its prologue. Our handler handles the system call and jumps back to the origin OS handler (*HandleContinue*)

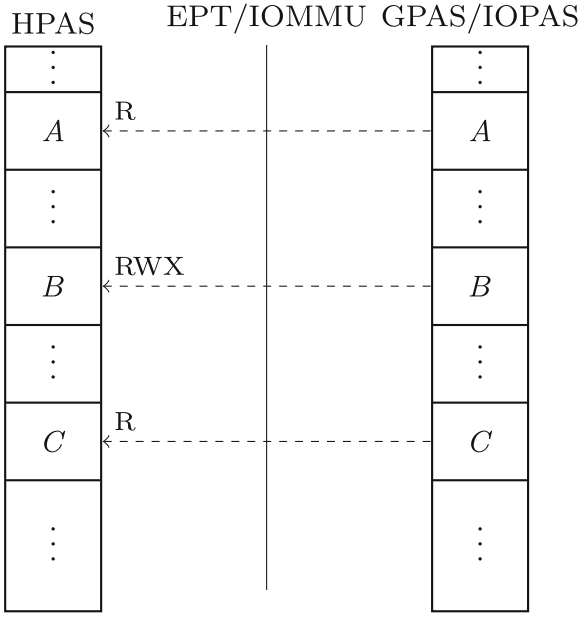

Fig. 5 Physical address space as perceived by the guest and I/O peripherals (right). Physical page *A* contains code/data of the hypervisor and therefore read-only rights. Physical page *B* contains code/data of the guest's OS and therefore has full rights (RWX). Physical page *C* contains code/data of the monitoring component and therefore has read-only rights

in turn triggers a VM-exit. In case of a write attempt, the hypervisor emulates the write instruction as if it had been done on the original page (initially, the original page is copied) and resumes the execution of the guest. In case of a read attempt, the hypervisor gives the guest the illusion that it reads from the original page. Figure 6 depicts this process. This technique is similar to a technique known as *invisible-breakpoints* and is described in (Deng et al. 2013).

Evaluation

This section consists of three parts. First, we discuss the overall performance impact of our system. Specifically, we determine how the overall performance is affected by (1) the hypervisor, (2) the monitoring component, and (3) the analysis of a process. Next, we discuss the fourth component of our system, the process behavior analyzer. Finally, we demonstrate how the process behavior analyzer can be used to analyze malware. All the experiments were performed in the following environment:

- CPU: Intel i7-7500 CPU @ 2.70GHz
- RAM: 16GB
- OS: Windows 7 SP1 x86

Empirical evaluation

In this experiment, we tested the system in three scenarios:

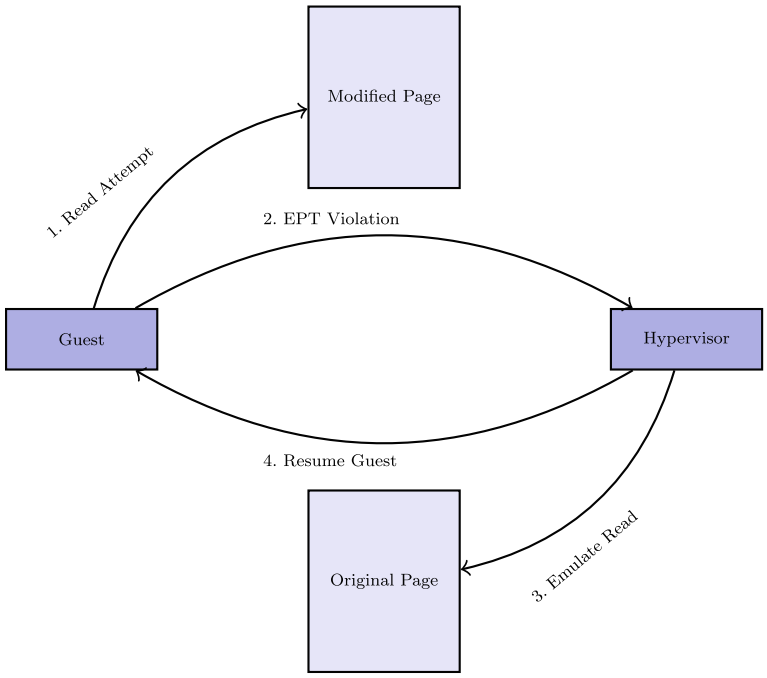

Fig. 6 Hypervisor memory modifications transparency. First, the guest attempts to read from the modified page. Because this page is configured to have only execute permissions, and an EPT violation occurs. Next, the hypervisor emulates the read instruction as if it had been done on the original memory page. Finally, the hypervisor resumes the execution of the guest

- without a hypervisor
- with a hypervisor and disabled monitoring component
- with a hypervisor and enabled monitoring component

We chose three benchmarking tools for Windows:

(a) PCMark 10 – Basic Edition
(b) PassMark Performance Test 9.0
(c) Novabench 4.0.3

Each tool performs several tests and displays a score for each.

As can be seen in the results reported in Fig. 7, the hypervisor degrades the performance by no more than 5%, and the monitoring component degrades the performance by, at most, an additional 1%.

We also checked the performance overhead of a monitored process. We selected an application that performs heavy usage of system services, the PassMark benchmark tool (the one described in the previous paragraph). We used a buffer size of 122k bytes and a regular file on the disk as the external source. The final dump file was 44MB in length. We did not observe any noticeable difference in performance.

Process behavior analyzer

The process behavior analyzer receives two parameters as input: (1) a sandbox configuration file, and (2) a dump file containing the recorded system calls. The process behavior analyzer reads the dump file and performs the following steps for each system call:

1. If the current system call has a configuration entry, the next data are fetched from the dump file according to the described configuration.
2. If the current system call has no configuration entry, the maximum possible number of arguments are fetched from the dump file. For a better analysis experience, the process behavior analyzer filters the unnecessary arguments (e.g., if *NtOpenFile* receives 6 arguments, the last 11 arguments are dropped).

The process behavior analyzer provides two JSON formatted files: (1) per-system call JSON file and (2) sequential system call JSON file.

The per-system call JSON file provides a list that contains only the system calls that have configuration entry in the sandbox configuration file. Each of the entries in the list contains a list of all occurrences of the specific system call, ordered chronologically. Each item in the list con-

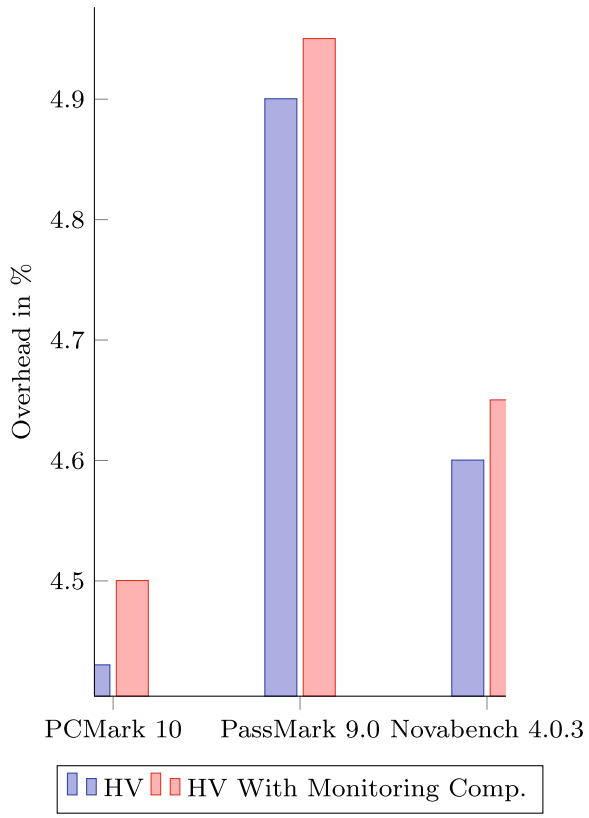

Fig. 7 Overhead of the benchmark execution under two conditions: (a) with HV only, and (b) with HV and enabled monitoring component

tains the information regarding the specific system call as described in the configuration file.

The sequential system call JSON file provides a list of all the conducted system calls (also for those that have no entry in the sandbox configuration file). The provided list is ordered chronologically. Each item in the list contains the parameters of the system call.

Malware analysis experiment

In this experiment, we selected a sophisticated malware (*W32.HfAdaware.4140*) from *das malwerk* Das Malwerk. Then, we set up a malware analysis lab and monitored it using our system. Finally, we analyzed the results with the process behavior analyzer. As described, the process behavior analyzer generates two output files. Figure 8 depicts the per-system call JSON report. Specifically, the *NtCreateUserProcess* system call is collapsed. As can be seen, the process behavior analyzer shows detailed information regarding the system call that has been conducted. For example, it shows the process' creation flags, image pathname, process arguments, and so on. As can be understood, the malware attempts to create a process from a temporary executable file (which it previously created) with a list of arguments (some of which are Base64 encoded). Figure 9 depicts the sequential JSON report. Specifically, three system calls are shown. The first system call is *NtCreateFile*, in which the malware attempts to create a temporary file named *dicabfcedb.zip*. After the *NtCreateFile* are two *NtWriteFile* system calls on which the malware attempts to write data to the created file. As

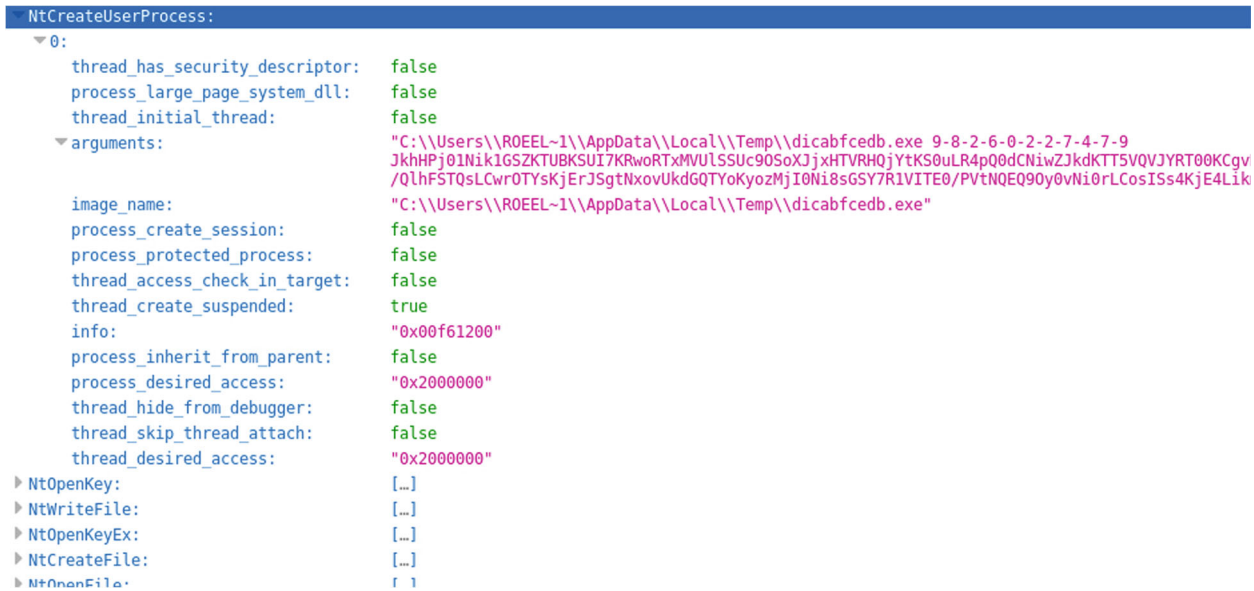

Fig. 8 *W32.HfsAdware.4140* results demonstration - per-system call JSON. An unknown process is created (*dicabfcedb.exe*). The process is given few parameters. Some of which seem to be Base64 encoded

Fig. 9 *W32.HfsAdware.4140* results demonstration - sequential JSON. System call #8447 creates a ZIP file. The following system calls perform the actual write to the file. System call #8449, writes, among others, the ZIP signature bytes (0x504b0304)

can be understood, the written data seems to be in the *ZIP* format.

Related work

The approaches for malware analysis can be divided into two broad categories: static and dynamic. Static methods form the foundation for popular antiviruses, which attempt to detect a particular pattern in the analyzed program's instructions. Due to the widely deployed evasion techniques, the dynamic analysis methods to which our approach belongs seem to be more robust against modern malware.

A recent survey on dynamic malware analysis (Or-Meir and et al. 2019) presents multiple classifications and comparisons between the current analysis techniques. In particular, Or-Meir et al. select the following foundations of dynamic analysis systems: (a) bare metal, (b) virtual machine, (c) hypervisor, (d) emulation, (e) volatile memory acquisition. The bare metal approach is too time-consuming to be used in large-scale scenarios. Volatile memory acquisition-based systems perform periodic snapshots, potentially missing malicious activity initiated and completed between subsequent snapshots.

Approaches (b)-(d) are more closely related to the current work. All these approaches share a common characteristic: they execute the malware in a simulated environment. The malware analyzer can reside either inside or outside the simulated environment. Analyzers that share the execution environment with the malware, like (Willems et al. 2007; Oh et al. 2010; Mohaisen et al. 2015; Bayer and et al. 2006), have direct access to the operating system services, thus enabling a more meaningful analysis. Unfortunately, such analyzers can be easily detected by the malware. External analyzers are less susceptible

to detection. Our approach has a component that executes inside the simulated environment. However, unlike previous solutions, our hypervisor hides the internal component from the potential malware, making it comparable with systems that use external analyzers while preserving the system's efficiency and access to operating system services.

The emulation-based methods construct the simulated environment in software without any assistance from the hardware. Unfortunately, perfect emulation is hard to achieve due to the complexity of the underlying hardware. Therefore, emulation-based methods, such as Anubis (Mandl et al. 2009) and Bitblaze (Song and et al. 2008), can be detected by malware, as was described by Lindorfer et al. (2011). Our approach employs hardware-assisted virtualization, thus making it resistant to such detection methods.

Virtualization solutions use two types of hypervisors: type I (Dinaburg et al. 2012; Lin et al. 2018), referred to as hypervisor-based solutions, and type II (Mohaisen et al. 2015; Cohen and Nissim 2018), referred to as virtual machine-based solutions. Type I hypervisors can operate directly over the hardware, whereas type II hypervisors require an operating system to mediate between the hardware and the hypervisor itself. Both cases use a full hypervisor, which runs at least one operating system acting as the malware's execution environment. Full hypervisors provide the virtual machines with emulated devices, communication channels, and other artifacts that evasive malware can detect (Afianian and et al. 2019). Our approach uses a thin hypervisor that provides its single virtual machine and its operating system with direct access to the hardware resources.

To the best of our knowledge, the only solution that uses a thin hypervisor is MAVMM (Nguyen et al. 2009). MAVMM does not use internal components to improve its undetectability, thus requiring MAVMM to rely on simple and slow output devices, like the serial port. Unlike MAVMM, our approach uses an internal component whose presence is hidden by the hypervisor. The internal component can utilize the operating system functionality to output the collected data to any external device. Besides, in our approach, the internal component intercepts the monitored events without requiring transitions to the hypervisor, thus improving the overall performance.

Timing-based evasion attacks on malware analyzers are challenging to circumvent. We note that using a thin hypervisor, as in our approach, keeps the time discrepancies at a minimum. Hypervisor detection and prevention (Algawi et al. 2019) is an arms race that is out of the current paper's scope. However, recently a method was proposed (Lin et al. 2018) to emulate local and remote time sources to manipulate malware's time perception. Our system can adopt this method to improve its transparency.

Limitations and future work

Kernel integrity

Our method uses a monitoring component that runs in the context of the guest's OS. We have seen that the recorded data is stored in a local buffer, which must be written to an external source once it becomes full. As the monitoring component is a kernel application, one may want to use existing implemented functionality (e.g., *ZwWriteFile*) to write the buffer to a local file. However, the threat model often assumes that the kernel code is not to be trusted. Our method can be extended with a kernel integrity verification method (Leon et al. 2018). Because the performance impact of the method described in (Leon et al. 2018) is negligible, we believe the combination with our method will not yield a significant performance loss.

Sandbox configuration automation

"Sandbox configuration file" section describes the sandbox configuration file. As of now, the malware analyst must manually fill in the system call numbers, parameters indexes, field offsets, and so on. Moreover, information such as the system call number may differ between each OS. This process can be almost fully automated. In Windows, a *Program Database (PDB)* file stores debugging-information regarding an executable file. A malware analyst could obtain the PDB of the kernel image of the system on which the monitored process will run (the *PDB* can be downloaded directly from Microsoft servers). A *PDB* parser tool could then extract the required information regarding each system call.

Detection

Modern malware try to detect the environment it runs on (Lindorfer et al. 2011). The malware detects sandboxes and hypervisors. On ARM there has been work on defeating hypervisor based malware detection (Petsas et al. 2014). We believe that it is possible to construct hypervisor that cannot be detected (Algawi et al. 2019) on x86 and x64 supporting our approach.

Conclusions

We present here a hypervisor-based system for dynamic malware analysis. Compared to current hypervisor-based systems, our system uses a novel approach in which a specially crafted monitoring component is injected into the address space of the guest OS. The hypervisor protects the monitoring component from detection and modification. We have shown that because the entire handling is done within the guest context, the performance overhead is negligible. In addition, we have seen that the hypervisor assures that the monitoring component remains completely transparent to the guest OS. The main limitation of our method is the transparency of our hypervisor. However, we believe that because the intervention of the

hypervisor is kept to a minimum, our method can be extended to also provide hypervisor transparency.

Acknowledgements
Not applicable

Authors' contributions
The ideas presented in this manuscript are based on discussions of all authors. Roee and Anat implemented the system. Michael wrote the first draft of this manuscript. All authors improved the manuscript. All authors read and approved the final manuscript.

Funding
This research was not funded.

Availability of data and materials
Please contact the correspondence author for the source code and the complete data presented here.

Competing interests
The authors' declare that they have no competing interests.

Author details
[1]Shenkar College, Ramat Gan, Israel. [2]Department of Software Engineering, Shamoon College of Engineering, Beer-Sheva, Israel. [3]College of Management Academic Studies, Rishon LeTsiyon Israel. [4]University of Jyväskylä, Jyväskylä, Finland.

Received: 26 October 2020 Accepted: 16 February 2021
Published online: 02 June 2021

References

Afianian A, et al. (2019) Malware dynamic analysis evasion techniques: A survey. ACM Comput Surv (CSUR) 52(6):1–28
Algawi A, Kiperberg M, Leon R, Resh A, Zaidenberg N (2019) Creating Modern Blue Pills and Red Pills. In: ECCWS 2019 18th European Conference on Cyber Warfare and Security. Academic Conferences and publishing limited, UK. p 6. https://jyx.jyu.fi/bitstream/handle/123456789/67098/1/CreatingModernBluePillsandRedPills.pdf
AMD (2010) AMD64 Architecture Programmer's Manual Volume 2: System Programming. https://www.amd.com/system/files/TechDocs/24593.pdf
ARM Ltd. (2013) ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile. https://developer.arm.com/documentation/ddi0487/latest
Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebauer R, Pratt I, Warfield A (2003) XEN and the art of virtualization. In: Proceedings of the 19th ACM Symposium on Operating System Principles. SOSSP, Bolton Landing
Basya D, Low R, Stamp M (2013) Structural entropy and metamorphic malware. J Comput Virol Hacking Tech 9(4):179–192
Bayer U, et al. (2006) Dynamic analysis of malicious code. J Comput Virol 2(1):67–77
Bayer U, Kruegel C, Kirda E (2005) TTanalyze: A Tool for Analyzing Malware. In: EICAR. pp 180–192. https://scholar.google.com/scholar?cites=7834276370136238241&as_sdt=2005&sciodt=0,5&hl=en
Ben Yehuda R, Shlingbaum E, Tayouri S, Gershfeld Y, Zaidenberg NJ (2021) Hypervisor Memory acquisition for ARM In Forensic Science International: Digital Investigation
Branco RR, Barbosa GN, Neto PD (2012) Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-VM techniques Vol. 1. pp 1–27. https://scholar.google.com/scholar?cluster=10089934970221990271&hl=en&as_sdt=2005&sciodt=0,5
Case A, Richard III GG (2017) Memory forensics: The path forward. Digit Investig 20:23–33
Christodorescu M, Jha S, Seshia SA, Song D, Bryant RE (2005) Semantics-aware malware detection. In: Proceedings of the 2005 IEEE Symposium on Security and Privacy (Oakland 2005), Oakland, CA, USA, May 2005. ACM Press, New York. pp 32–46
Cohen M (2014) Robust Linux Memory Acquisition with Minimal Target Impact. Johannes Stuettgen and Michael Cohen. The proceedings of The Digital Forensic Research Conference DFRWS 2014 EU, Amsterdam
Cohen A, Nissim N (2018) Trusted detection of ransomware in a private cloud using machine learning methods leveraging meta-features from volatile memory. Expert Syst Appl 102:158–178
D'Elia DC, Coppa E, Palmaro F, Cavallaro L (2020) On the Dissection of Evasive Malware. IEEE Trans Inf Forensic Secur 15:2750–2765
Das Malwerk Malware Samples. https://dasmalwerk.eu/, https://das-malwerk.herokuapp.com/about. Accessed Sept 2019
Deng Z, Xu D, Zhang X, Jiang X (2012) Introlib: Efficient and transparent library call introspection for malware forensics. Digit Investig 9:S13–S23
Dinaburg A, Royal P, SHARIF MI, LEE W (2012) Ether: malware analysis via hardware virtualization extensions. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS). IEEE. pp 51–62. https://ieeexplore.ieee.org/document/6329220
Ding B, Wu Y, He Y, Tian S, Guan B, Wu G (2012) Return-oriented programming attack on the xen hypervisor. In: 2012 Seventh International Conference on Availability, Reliability and Security. IEEE, Prague. pp 479–484. https://doi.org/10.1109/ARES.2012.16
Deng Z, Zhang X, Xu D (2013) Spider: Stealthy binary program instrumentation and debugging via hardware virtualization. In: Proceedings of the 29th Annual Computer Security Applications Conference, ACSAC 13. pp 289–298
Egele M, Scholte T, Kirda E, Kruegel C (2008) A survey on automated dynamic malware-analysis techniques and tools. ACM Comput Surv (CSUR) 44(2):1–42
Feng Y, Anand S, Dillig I, Aiken A (2014) Apposcopy: Semantic-Based detection of android malware through static analysis. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014). pp 576–587. https://dl.acm.org/doi/abs/10.1145/2635868.2635869?casa_token=Cd0d2xUKguYAAAAA:g-q0Fjc4W0JxkZ8nd3aqV-2kpnwiIJwLQjMofj0nILNaKOymYZVJ3BaqgXITEdrWAMRvXS9kDRJh
Ferrie P (2006) Attacks on virtual machine emulators. Symantec Adv Threat Res 5:1–3. https://scholar.google.com/scholar?cluster=490094371751728691&hl=en&as_sdt=2005&sciodt=0,5
Franklin J, Luk M, McCune JM, Seshadri A, Perrig A, van Doorn L (2008) Remote Detection of Virtual Machine Monitors with Fuzzy Benchmarking, Vol. 42. Association for Computing Machinery, New York. https://doi.org/10.1145/1368506.1368518
Gandotra E, Bansal D, Sofat S (2014) Malware analysis and classification: A survey. J Inf Secur 5(02):56
Garfinkel T, Adams K, Warfield A, Franklin J (2007) Compatibility Is Not Transparency: VMM detection myths and realities. In: USENIX Workshop on Hot Topics in Operating Systems (HotOS). USENIX Association, USA. pp 1–6
Guarnieri C, Fernandes D (2010) Cuckoo Automated Malware Analysis System. http://www.cuckoobox.org/
Hull G, John H, Arief B (2019) Ransomware deployment methods and analysis: views from a predictive model and human responses. Crime Sci 8(1):2
Intel Corporation (2007) Intel 64 and IA-32 Architectures Software Developer's Manual, vol. 3. https://software.intel.com/content/dam/develop/external/us/en/documents-tps/253668-sdm-vol-3a.pdf
Kharaz A, Arshad S, Mulliner C, Robertson W, Kirda E (2016) UNVEIL: A large-scale, automated approach to detecting ransomware. In: 25th USENIX Security Symposium (USENIX Security 16). USENIX Association, Austin. pp 757–772
Kinder J, Katzenbeisser S, Schallhart C, Veith H (2005) Detecting malicious code by model checking. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, Berlin, Heidelberg. pp 174–187. https://www.springer.com/gp/book/9783540266136
Kirat D, Vigna G, Kruegel C (2011) BareBox: efficient malware analysis on bare-metal. In: ACSAC. ACM, New York. https://doi.org/10.1145/2076732.2076790
Kirat D, Vigna G, Kruegel C (2014) Barecloud: bare-metal analysis-based evasive malware detection. In: USENIX Security, San Diego. pp 287–301. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/kirat
Lengyel TK, Maresca S, Payne BD, Webster GD, Vogl S, Kiayias A (2014) Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system. In: Proceedings of the 30th Annual Computer Security Applications Conference on ASAC '14. pp 386–395. https://dl.acm.org/doi/abs/10.1145/2664243.2664252
Leon RS, Kiperberg M, Zabag Leon AA, Resh A, Algawi A, Zaidenberg N (2018) Hypervisor-Based Whitelisting of Executables. IEEE Sec Priv 17(5):58–67. https://doi.org/10.1109/MSEC.2019.2910218

Li J, Wang Q, Jayasinghe D, Park J, Zhu T, Pu C (2013) Performance Overhead Among Three Hypervisors: An Experimental Study using Hadoop Benchmarks. IEEE Int Conf Big Data. 9–16. https://doi.org/10.1109/BigData.Congress.2013.11
Lin C-H, Pao H-K, Liao J-W (2018) Efficient dynamic malware analysis using virtual time control mechanics. Comput Secur 73:359–373
Lindorfer M, Kolbitsch C, Comparetti PM (2011) Detecting environment-sensitive malware. In: International Workshop on Recent Advances in Intrusion Detection. Springer, Berlin, Heidelberg. pp 338–357
Mandl T, Bayer U, Nentwich F (2009) ANUBIS ANalyzing unknown BInarieS the automatic way. In: Virus bulletin conference Vol. 1. p 2
Mohaisen A, Alrawi O, Mohaisen M (2015) AMAL: high-fidelity, behavior-based automated malware analysis and classification. Comput Secur 52:251–266
Moser A, Kruegel C, Kirda E (2007) Limits of static analysis for malware detection. In: Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007). IEEE. pp 421–430. https://doi.org/10.1109/ACSAC.2007.21. https://ieeexplore.ieee.org/abstract/document/4413008
Murray DG, Milos G, Hand S (2008) Improving Xen security through disaggregation. In: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments. pp 151–160. https://dl.acm.org/doi/abs/10.1145/1346256.1346278
Nguyen AM, Schear N, Jung H, Godiyal A, King ST, Nguyen HD (2009) Mavmm: Lightweight and purpose built vmm for malware analysis. In: 2009 Annual Computer Security Applications Conference. IEEE, Honolulu. pp 441–450. https://doi.org/10.1109/ACSAC.2009.48
O'Kane P, Sezer S, McLaughlin M (2011) Obfuscation: the hidden malware. IEEE Secur Priv 9(5):41–47
Oh J, Im C, Jeong H (2010) A system for analyzing advance bot behavior. In: International Conference on Information Systems, Technology and Management. Springer, Berlin, Heidelberg
Or-Meir O, et al. (2019) Dynamic malware analysis in the modern era—A state of the art survey. ACM Comput Surv (CSUR) 52(5):1–48
Petsas T, Voyatzis G, Athanasopoulos E, Polychronakis M, Ioannidis S (2014) Rage against the virtual machine: hindering dynamic analysis of android malware. In: Proceedings of the Seventh European Workshop on System Security. pp 1–6. https://dl.acm.org/doi/abs/10.1145/2592791.2592796
Rad B, Masrom M, Ibrahim S (2012) Camouflage in Malware: From Encryption to Metamorphism. Int J Comput Sci Netw Secur 12:74–83
Raffetseder T, Kruegel C, Kirda E (2007) Detecting System Emulators. In: International Conference on Information Security. Springer, Berlin. pp 1–18. https://link.springer.com/chapter/10.1007/978-3-540-75496-1_1
Roesch M (1999) Snort: Lightweight intrusion detection for networks. In: Lisa, Vol. 99, No. 1. pp 229–238. https://www.usenix.org/legacy/publications/library/proceedings/lisa99/roesch.html. https://www.usenix.org/legacy/publications/library/proceedings/lisa99/full_papers/roesch/roesch.pdf
Saeed IA, Selamat A, Abuagoub AMA (2013) A Survey on Malware and Malware Detection Systems. Int J Comput Appl 67(16):25–31
Shah SAR, Issac B (2018) Performance comparison of intrusion detection systems and application of machine learning to Snort system. Futur Gener Comput Syst 80:157–170
Song D, et al. (2008) BitBlaze: A new approach to computer security via binary analysis. In: International Conference on Information Systems Security. Springer, Berlin, Heidelberg
Sylve J (2012) Android Mind Reading: Memory Acquisition and Analysis with DMD and Volatility Shmoocon 2012. https://www.youtube.com/watch?v=oWkOyphlmM8. Accessed 25 Feb 2021
Ucci D, Aniello L, Baldoni R (2019) Survey of machine learning techniques for malware analysis. Comput Secur 81:123–147
Vidas T, Christin N (2014) Evading Android runtime analysis via sandbox detection. In: Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (ASIACCS '14). Association for Computing Machinery, New York. pp 447–458. https://dl.acm.org/doi/10.1145/2590296.2590325
Vinod P, Laxmi R, Gaur M (2009) Survey on Malware Detection Methods. In: Proceedings of the 3rd Hackers' Workshop on computer and internet security (IITKHACK'09). pp 74–79
VMWare (2005) VMware ESX server virtual infrastructure node evaluator's guide. https://www.VMware.com/pdf/esx_vin_eval.pdf. Accessed Sept 2019
White JS, Fitzsimmons T, Matthews JN (2013) Quantitative analysis of intrusion detection systems: Snort and Suricata. In: Cyber sensing 2013, vol. 8757. International Society for Optics and Photonics. p 875704. https://doi.org/10.1117/12.2015616
Willems C, Holz T, Freiling F (2007) CWSandbox: Towards automated dynamic binary analysis. IEEE Secur Priv 5(2):32–39. https://doi.org/10.1109/MSP.2007.45
Yalew SD, McGuire G, Haridi S, Correia M (2017) T2Droid: A TrustZone-based dynamic analyser for Android applications. In: 2017 IEEE Trustcom/BigDataSE/ICESS. IEEE. pp 240-247. https://ieeexplore.ieee.org/abstract/document/8029446
Yokoyama A, Ishii K, Tanabe R, Papa Y, Yoshioka K, Matsumoto T, Kasama T, Inoue D, Brengel M, Backes M, Rossow C (2016) SandPrint: Fingerprinting malware sandboxes to provide intelligence for sandbox evasion. In: International Symposium on Research in Attacks, Intrusions, and Defenses. Springer, Cham. pp 165–187
You I, Yim K (2010) Malware Obfuscation Techniques: A Brief Survey. In: 2010 International conference on broadband, wireless computing, communication and applications. IEEE. pp 297–300. https://doi.org/10.1109/BWCCA.2010.85. https://ieeexplore.ieee.org/abstract/document/5633410
Zaidenberg NJ (2018) Hardware rooted security in industry 4.0 systems, Vol. 51. IOS Press BV, Netherlands. http://ebooks.iospress.nl/volumearticle/50292
Zaidenberg NJ, Kiperberg M, Yehuda RB, Leon R, Algawi A, Resh A (2020) Hypervisor Memory Introspection and Hypervisor Based Malware Honeypot. In: Mori P, Furnell S, Camp O (eds). Information Systems Security and Privacy. ICISSP 2019. Communications in Computer and Information Science, vol 1221. Springer, Cham. https://doi.org/10.1007/978-3-030-49443-8_15
Zaidenberg N, Neittaanmäki P, Kiperberg M, Resh A (2015) Trusted computing and drm. In: Cyber Security: Analytics, Technology and Automation. Springer, Cham. pp 205–212

Publisher's Note