Artifact Mitigation in High-Fidelity Hypervisors

Christopher Norine
Naval Postgraduate School
christopher.norine@nps.edu

Alan Shaffer
Naval Postgraduate School
alan.shaffer@nps.edu

Gurminder Singh
Naval Postgraduate School
gsingh@nps.edu

Abstract

The use of hypervisors for cyber operations has increased significantly over the past decade, resulting in a concomitant increase in the demand for higher fidelity hypervisors that do not exhibit the markers, or artifacts that identify the execution platform type (virtualized or bare metal), prevalent in most currently available virtualization solutions. To address this need, we present an in-depth examination of a specific subset of virtualization artifacts in order to design and implement a method of mitigation that reduces the detectability of these artifacts. Our analysis compares the performance of a bare metal machine, a virtual machine without artifact mitigation, and a virtual machine with our proof-of-concept mitigation technique applied to a temperature sensor. Results of the implementation are analyzed to determine the potential impact on system performance and whether our mitigation technique is appropriate for extending high-fidelity hypervisors.

1. Introduction

High-fidelity hypervisors are becoming a required asset in the cyber operations community. From malware analysis and honeypot operations to training environments for testing cutting-edge cyber tools and techniques, virtualization offers a safe and isolated environment within which to research and test new methods. A downside to operating within virtual machines is that they often lead to artifacts (or markers) that, upon discovery, may allow an observer to realize they are not operating on a bare metal machine. While most of these artifacts are a byproduct of tighter host-guest integration and proper separation between host machine and guest virtual machine, there may be a desire to hide or mitigate these artifacts.

First, malware analysis can greatly benefit from a high-fidelity hypervisor. Dinaburg et al. described how "malware authors are incentivized to complicate attempts at understanding the internal workings of their creation" [1]. These complications include techniques that can be described as anti-debugging, anti-instrumentation, and anti-VM to frustrate would-be analysts and prevent deeper understanding of the malware. Indeed, Chen et al. characterized the prevalence of evasion techniques in modern malware. According to their research, over 40% of the 6,900 total malware samples they examined reduced their malicious behavior whenever a debugger was attached, or when the malware suspected it was executing within a virtual machine [2]. Artifact mitigation enables a high-fidelity hypervisor to show no signs of its virtualized environment, allowing analysts to more fully explore the functionality of target malware.

Second, an organization running a honeypot would benefit greatly from a high-fidelity hypervisor. A virtualized environment is ideal for the execution of a honeypot, therefore malware that encounters such a system will likely attempt to determine if the environment is virtualized or not [2]. A high-fidelity hypervisor with artifact mitigation would be a better environment for honeypots to operate in, as they would exhibit the behaviors of a bare metal machine without any of the artifacts typically present in virtual machines.

Lastly, it is essential for cyber operators to have a holistic environment in which to develop, test, train and rehearse their cyber tools and techniques. From an offensive standpoint, it would be impractical to test offensive cyber operations (OCO) on a bare metal machine, since the results will likely damage or corrupt these test systems. Recovery will ultimately take time away from the cyber operators, and either reduce the total time spent training and testing or increase the time it takes to reach a working solution. Either scenario is understandably not ideal or desirable. By offering a high-fidelity hypervisor that is able to present a system's "digital twin," we are able to suppress the artifacts that affect feedback to the operator while still providing a target environment that behaves *exactly* as its bare metal equivalent would.

For these reasons, it is essential to examine the different aspects that reduce the fidelity of an off-the-shelf hypervisor. In doing so, we attempt to design and implement mitigation measures that are able to increase the overall fidelity of hypervisors, while ensuring that

URI: https://hdl.handle.net/10125/71466
978-0-9981331-4-0

execution of the hypervisors and guest operating systems are not compromised.

The key contributions of this work include:

1. Design and implementation of **smokescreen**, a DRAKVUF plugin used to mitigate virtualization artifacts.
2. Implementation of a test framework to analyze the performance and mitigation capabilities of virtualized environments with and without the new plugin, as compared to a bare metal system.

2. Background and Related Work

The goal of a high-fidelity hypervisor is to present a virtualized machine that is indistinguishable from a physical machine, or a "digital twin" of sorts. The aim here is to improve upon the equivalence property of virtual machines described by Popek and Goldberg [3]. It is important to point out that our goal is a higher fidelity *virtualized* experience, not a more accurate *emulation* experience. Emulation is defined by Mallach as "a process whereby one computer is set up to permit execution of programs written for another computer" [4].

With emulation, translation of the program to be executed is required, which is not a requirement for virtualization. Previous research into high-fidelity hypervisors has included an approach by Zhang, Xie, Dong, Yang, and Zhou, who were able to synchronize the operations of a virtualized machine and a physical component emulator to create a more holistic virtualized environment [5]. Although primarily intended to show that cyber physical system (CPS) software controllers do not require a physical setup to be able to test and develop, it presents a potential avenue to increase the fidelity of currently available hypervisor solutions without sacrificing performance and tipping off observers to the presence of a hypervisor.

2.1. Hypervisor Overview

Hypervisors are designed to run virtual machines with low overhead. Typical hypervisors can operate on a single machine or can utilize cloud or other distributed resources to support many virtual machines that operate concurrently. Hypervisors may run as a layer between hardware and guest operating system (a Type 1 hypervisor), or as a user-level application running on another operating system (a Type 2 hypervisor). An example of a Type 1 hypervisor is the Xen Hypervisor, and an example of a Type 2 hypervisor is VMware Workstation.

A Type 1 hypervisor maintains full control of the host system and its resources. Since these hypervisors occupy the same level as an operating system, they typically act in both capacities as virtual machine monitor for VMs, and as operating system for the host. For this reason, Type 1 hypervisors tend to enjoy lower amounts of overhead, as compared to their Type 2 counterparts.

A Type 2 hypervisor has full control of the host CPU during VM execution. Compared to Type 1 hypervisors, Type 2 hypervisors have additional overhead incurred as the host OS and hypervisor execute switches, similar to CPU context switches, to achieve virtualization [6].

In addition to hypervisor types, virtualization can also be categorized into binary translation, full virtualization, and paravirtualization.

Binary translation is a form of recompilation where instructions are translated from the source instruction set to a target instruction set (i.e., x86 machine instructions are translated into the equivalent ARM machine instructions). This is different from emulation, where the translation is simply executing the target instructions by utilizing a corresponding set of instructions from the source instruction set [4]. This translation can either be done statically (where recompilation occurs prior to runtime) or dynamically (where program instructions are translated as they are read). This method of virtualization is generally considered difficult since a translator program tends to be very specialized and a significant amount of extra work tends to be required to extend the translator to support new target instruction sets [7].

Full virtualization (also known as hardware virtualization) involves the guest system being completely unaware of the presence of the hypervisor. Instructions that are sensitive or privileged must be caught by the hypervisor without being observable by the guest OS. Although full virtualization does not require any specialized instruction sets or device drivers within the guest OS, it can incur significant performance penalties as the CPU executes context switches to allow the hypervisor to handle sensitive or privileged instructions.

Paravirtualization (PV) is a virtualization method where the guest OS is fully aware of the hypervisor. The guest OS contains a specialized kernel and other virtualization-aware device drivers that can take advantage of the communication channels that typically exist between the hypervisor and VMs. Although PV guests require less overhead due to utilization of the communication channels between hypervisor and VM, it comes at the cost of reduced security, as vulnerabilities in PV drivers can expose the hypervisor to an attacker [8], and the added requirement for specialized drivers.

2.2. Hypervisor Artifacts

Virtualization artifacts are markers or indicators of either the presence of a VMM, or of the fact that a system is virtualized. Most of these artifacts fall within one of three categories: service, process, or file system artifacts; memory artifacts; and virtualization-specific artifacts, which are further broken down into hardware-specific and capability-specific artifacts.

Service, process, or file system artifacts are exemplified by paravirtualization-specific drivers. By utilizing paravirtualization and the hypervisor-guest communication channels, it is possible to greatly reduce the performance overhead of a hypervisor, making a strong case for their use. However, for ease of use most hypervisors are transparent about paravirtualization drivers, causing many artifacts within the guest itself. For example, a VMware Workstation guest running Windows XP with PV-aware drivers installed has over 50 references to "VMware" in the file system and over 300 references in the registry [9].

Memory artifacts tend to involve references to the hypervisor itself within the system memory. This can be observed in the way that the operating system stores essential tables and references. For example, the Interrupt Descriptor Table (IDT) is a kernel structure that holds a list of pointers to the operating system interrupts [9]. Since the hypervisor and guest OS both contain and maintain their own IDT they cannot be located at the same physical memory address, so tools such as The Red Pill [10] that are able to inspect the IDT pointer can determine if a machine is virtualized or not.

Virtualization-specific artifacts typically result from the fact that most hypervisors are transparent with their paravirtualization drivers within a guest OS. On a Linux guest, within locations such as system logs or the various virtual file system directories (e.g., /proc or /sys), or from the output of commands such as "dmesg", it is possible to discover references to the hypervisor [9]. This type of artifact can also be caused by the extended instruction sets typically seen in PV guests. For example, VMware and Xen add additional machine instructions to the processor instruction set that allow usage of the hypervisor-guest communication channel. Tools such as VMDetect will intentionally execute these instructions with the goal of detecting whether execution was successful or not, where success would indicate that the system was virtualized [9].

2.3. Hypervisor Detection

Current hypervisor detection techniques include *count-based detection* and *register inspection-based detection*. Research at the University of Minnesota quantified timing artifacts within virtual machines. Thompson et al. discovered that by executing a loop of NOP instructions alongside a loop of CPUID instructions (a privileged instruction requiring hypervisor intervention) and comparing the ratio of total instructions executed, detectable differences were observable between virtual and bare metal machines [11]. This is due to the additional latency introduced by the hypervisor trapping these sensitive instructions for handling before returning control to the guest [12].

Another method of hypervisor detection exploits the presence of sensitive but unprivileged instructions or instruction set extensions like those implemented in PV guests. The Red Pill [10] and Paranoid Fish [13] both utilize the SIDT (Store Interrupt Descriptor Table register) instruction to discover the hypervisor, while software tools like ScoopyNG [14] and VMDetect [9] exploit the presence of additional instruction sets to expose the hypervisor.

2.4. Detection Mitigation

Current detection mitigation techniques and software tools tend to center around modification of the hypervisor configuration. Liston et al. discovered the existence of undocumented configuration options for VMware hypervisors that, when set a certain way, would effectively "break" the hypervisor-guest communication channels and defeat both The Red Pill and ScoopyNG [9]. The downside to this technique is that these configuration options are not documented or officially supported, meaning there is no guarantee that a software update will not render the mitigation technique ineffective.

Another mitigation method uses a proof-of-concept application called VMmutate to obfuscate the presence of a VMware hypervisor through modification of the VMX configuration parameters. The tool will either modify or disable the VMware "magic value" that must be present to utilize the extended instruction set to communicate with the hypervisor. These mitigation techniques were found to be enough to defeat The Red Pill as well as portions of the ScoopyNG suite [8].

The downside to these techniques is that they tend to require extensive hypervisor configuration or virtual machine memory alterations. Either of these can lead to less portable solutions as the hypervisor and PV tools/drivers are modified.

3. System Design

For our research, we decided on an appropriate hypervisor and targeted guest system by examining the most commonly used hypervisors and operating systems being run by web servers on the internet. According to W3Techs, most web servers utilize a Unix-based

operating system, with Ubuntu being the most common [15]. Therefore, we chose to use Ubuntu 16.04.6 LTS as it is still within its support window through the maintainer, Canonical, and is also likely to continue being used "in the wild."

We elected to use a Type 1 hypervisor over an application-based Type 2 hypervisor as the former is more common in commercial environments. Specifically, the open source Xen hypervisor was chosen due to its wide adoption within the tech sector. Additionally, we used the DRAKVUF software suite, which was originally built to work alongside the Xen hypervisor, allowing us to leverage that capability to achieve a working solution faster and more efficiently.

3.1. DRAKVUF

DRAKVUF is a software suite designed as a "virtualization based agentless black-box binary analysis system" [16]. Although originally developed for malware analysis, we found that by utilizing specific features of DRAKVUF, we could attempt to mitigate artifacts found in VMs. The tool allowed us to both examine and manipulate execution of binaries within a VM without having to install any additional analysis software with the guest host.

3.2. LibVMI

The XenAccess Project was originally conceived to be a Xen hypervisor-specific solution for virtual machine introspection. From this project, LibVMI was derived as an offshoot meant to be more platform-agnostic and able to support different hypervisors. For our research, it provided the ability to monitor (by reading VM memory) and control (by writing VM memory) from outside the guest VM, allowing us to remain undetected from the VM's perspective [17]. Utilizing LibVMI with the Rekall profile (discussed next) we were able to bypass the oft-used KdDebuggerData (KDBG) structure, which may be corrupted or encoded depending on the guest operating system.

3.3. Rekall

Rekall is a Python-based open source framework that is used for "extraction and analysis of digital artifacts (in) computer systems" [18]. It functions by analyzing the currently running kernel to determine its configuration, and then uses that configuration along with the kernel source headers to catalogue the locations of kernel structures. That catalogue is output as a standardized JSON (JavaScript Object Notation) file for use by DRAKVUF.

This tailored catalogue file provides important information such as the base address of the kernel structure itself, and additionally provides address offset of various kernel memory objects and system call locations. By utilizing these locations, we can leverage DRAKVUF and LibVMI to trap any system calls we determine necessary without any modification to the guest VM or its kernel.

3.4. DRAKVUF Plugins

The last essential component in our design is the powerful plugin system that is built into DRAKVUF. We developed and integrated a plugin called **smokescreen** that monitors VM execution and traps any system calls that would attempt to execute artifact-exposing binaries. If any call is trapped, **smokescreen** replaces the pathname of the binary to be executed with a path to a modified version that would suppress the artifacts and generate a reasonable output. Details of the **smokescreen** implementation are provided below.

4. System Implementation

We investigated the possibility of mitigating device and capability artifacts that arise from a VM not having access to host hardware devices, such as temperature sensors found on modern computers. While many sensor devices are emulated by the hypervisor, this is not always the case. For example, a stock Ubuntu 16.04.6 LTS installation operating as a fully virtualized guest with the Xen hypervisor does not have access to the temperature sensor normally found in a CPU, so attempts to query those sensors are typically met with an error or failure condition. This also can include other system sensors such as light or motion sensors that also are not be exposed to the VM.

4.1. Hardware Component Emulation

We chose to implement a temperature sensor found on an Intel Core i7-6700 CPU due to its common usage. Research of the product specifications and datasheet for this CPU was done to ascertain whether a model for the temperature sensor could be found, or if one could potentially be derived from other specifications. References to T_{CASE} describe it as "the maximum temperature allowed at the processor Integrated Heat Spreader (IHS)" and corresponds to a linear model where the maximum temperature was a function of the power utilization at a particular point [19]. We used this linear model to estimate a potential maximum CPU

temperature within a VM utilizing only current processor utilization statistics, which *are* available in a guest VM.

4.2. Software Component

The system's software component primarily consists of a modified version of the Linux *sensors* program, which is part of the *lm-sensors* software toolkit. The unmodified program, shown in Figure 1, outputs data from the contents of various files in the system's virtual file systems that contain the raw output values of system hardware sensors. The *sensors* program reads this file data, formats it as appropriate, and then outputs it for a user process. However, in a virtualized environment one is more likely to encounter an error condition, as those sensors are not exposed to the VM and are thus unreadable, exposing an artifact of virtualization.

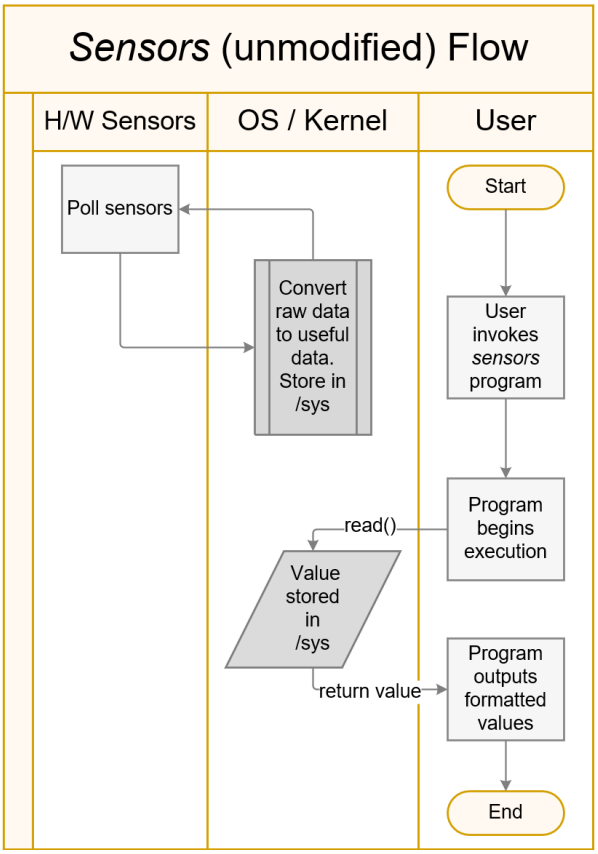

Figure 1. Unmodified *sensors* execution flow

We modified the *sensors* binary to mitigate the lack of sensors being exposed to the VM (a capability/device artifact). The modified binary would suppress the error condition where no sensor data was found and replace it with a lookup function that estimates the desired sensor data. While a simple approach would have been to have the program output a static value at runtime, for example an average temperature across most workplace environments, our goal of increasing the hypervisor fidelity meant that the output should reasonably reflect the current state of the apparent bare metal machine and its environment.

To this end, we implemented a library-type function call that, when invoked, would sample the current utilization statistics within the VM and execute a lookup of the appropriate T_{CASE} temperature value. This modified flow is shown in Figure 2. System utilization was determined by querying the */proc/loadavg* file, which outputs a 1-, 5-, and 10-minute running average of CPU load across all cores as a floating-point value. By utilizing the 1-minute average as an instantaneous approximation of current system utilization, and the 5- and 10-minute averages as the "effect" of a cooling solution (e.g., the higher the long term utilization, the more likely the cooling fans have also been working to manage temperature for a longer period of time), we were able to implement a simple lookup function that scaled with system utilization. The result of this lookup was the value returned by *sensors*.

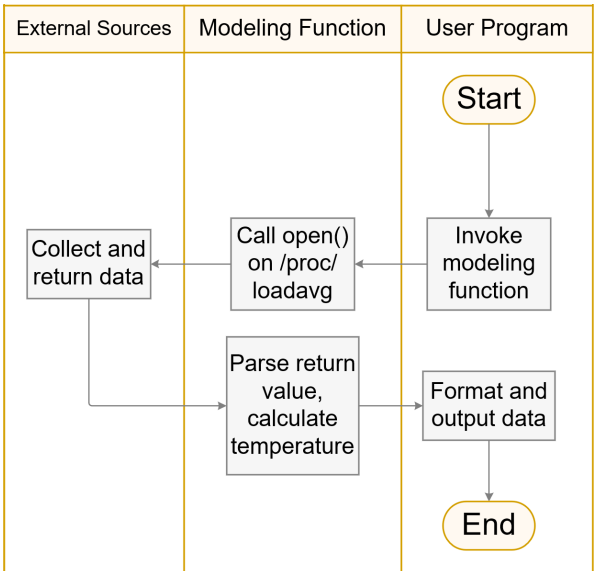

Figure 2. Modified *sensors* execution flow

4.3. Means of Implementation

We developed **smokescreen** as a plugin for DRAKVUF (implemented in C/C++) to monitor VM execution and trap any system calls that would attempt to execute artifact-exposing binaries. If any call is trapped, **smokescreen** will replace the pathname of the binary to be executed with a path to a modified binary that would suppress the artifacts and output a reasonable estimated output value.

To understand how **smokescreen** is able to modify pathnames without VM corruption, knowledge of 64-bit Sysv calling conventions was also required. Since the introspection and modification actions taken are executed at a low-level, they require knowledge of CPU registers and their memory values for any kernel system call on which we wish to perform introspection.

Our plugin itself was implemented as an extension of the base *plugin* class (per the specification) and consists of a total of seven files. Two of the files consist of the plugin itself, one is a header file containing our temperature estimation function, a patch file for the source code of *sensors*, two files that describe the system calls and file paths to be modified respectively, and finally a recompiled (i.e. modified) version of *sensors* that has our patch applied. The plugin files consist of *smokescreen.cpp* and *smokescreen.h* per plugin specification. The temperature lookup function is a single file, *temp_lib.h*, and is required to be co-located within the *sensors/lib* folder of the lm-sensors source code. This ensures that our patched *sensors* binary can compile correctly. The patch file must be applied to *main.c* of the *sensors* binary prior to compilation to ensure the function residing in *temp_lib.h* is executed correctly and prevents the leakage of VM artifacts.

The system call list is passed as a command line argument to DRAKVUF (via the *-S* argument) which can be either a relative or absolute path, while the list of binaries is located per the plugin initialization source code (for our research, it is at */home/xen/xen/binary_list.txt*). This location is modifiable within the *smokescreen.cpp* file as well. Lastly, the modified *sensors* binary must be located at a configuration-dependent path (in our case, */fake/usr/bin* on the guest VM) but this location is also easily modifiable within the *smokescreen.cpp* source file.

For the actual implementation, we found we needed to trap the *execve()* kernel system call. This system call requires three parameters: a pointer to the fully qualified path name as a string, a pointer to an argument vector, and a pointer to an environmental variable vector. According to AMD64 Sysv calling conventions, these parameters are found in the registers RDI, RSI, and RDX, respectively. For the purposes of our modifications, the argument and environmental variable vectors are not required to be modified for **smokescreen** execution. Since modification of the argument vector (i.e. command line arguments passed to *sensors* at execution time) would likely expose our modifications, it is best to leave it untouched which will ensure that our modified version will preserve functionality not directly related to our work, such as a version or help message. Similarly, the program does not utilize the environmental variables directly and modifying them may have unintended side effects during execution, so it is best left unmodified.

The plugin execution flow, shown in Figure 3, follows five steps:

1. Retrieve and lock (pause) the instance of the monitored VM.
2. Identify the pathname of the binary to be executed through VM introspection.
3. Compare the pathname to our predefined list of binaries that require redirected execution.
4. If a match is found, extract and modify the pathname to be the path to our *modified* binary. If not, execution of the VM will be resumed immediately with no modifications made.
5. Release (un-pause) the VM and allow execution to resume.

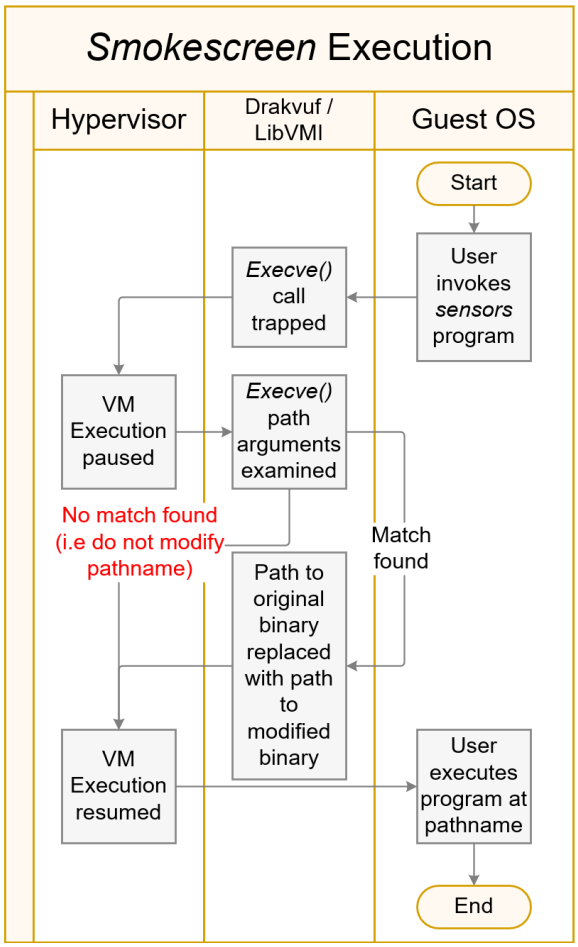

Figure 3. Smokescreen execution flow

During step 1, a reference to the VM instance allowed us to access to the values stored within the CPU registers as well as the contents of the VM's memory through virtual and physical memory addresses. As we are concerned with the *execve()* system call, it is

important that we are able to extract the pathname from the VM's RDI register for the given (caller) process. As DRAKVUF (via LibVMI) traps these calls, **smokescreen's** callback function is executed and able to query LibVMI for the VM's *vmi_instance_t* struct. This struct, in conjunction with the *drakvuf_trap_info_t* struct passed to the callback, allows us to gather important information such as the caller's PID (Process Identification number) and a complete snapshot of the CPU's registers and their values. When the system call we are waiting for is executed, RDI contains a pointer to a string representation of the file path to be executed.

Step 2 is executed via LibVMI's *vmi_read_str_va()* function. **Smokescreen** accomplishes this by asking for the contents of the RDI register for a specific VM's PID. This call returns to us a pointer to the string containing the fully qualified pathname of the binary to be executed.

Step 3 is done by taking the extracted pathname and executing a string comparison to our predefined list of binaries (which is established at plugin initialization). From there, step 4 branches into two possibilities. If a match is found, we construct a new pathname which leads to our modified binary and proceed to write the new string back to the original memory location referenced in the VM's RDI register. If no match is discovered, the VM is immediately released and execution continues with no modification to the register or memory contents.

Finally, we release the *vmi_instance_t* struct, which allows execution to continue within the VM. Unless the user of the guest VM is meticulously monitoring execution of all binaries on the system, it is unlikely that this process will be observed within the system in real time.

In summary, we have designed and implemented a system that can replace execution of binaries that could potentially leak the presence of a hypervisor with little impact to the guest VM. By placing modified binaries within the VM and utilizing VM introspection, we are able to successfully redirect execution and obfuscate the presence of our modified binaries, as the guest system still believes it is executing the original (un-altered) files. Therefore, even if a user were to examine the binaries they *believe* are being executed, there is no outward indication of any issue, since the unmodified files contain no markers or indicators that they are not actually the files being executed. Although we are forced to introduce file system artifacts (through the mere presence of the modified binaries within the guest VM), there is no requirement within the implementation of **smokescreen** to follow any naming convention that may enable easy identification, thus further obfuscating their presence on the guest VM.

5. System Testing

To test our system's ability to achieve artifact mitigation, we needed to examine the detectability of VM introspection when redirecting the guest system's execution. To this end, we examined the runtime of the modified *sensors* program in three environments: a bare metal machine, a VM where no introspection occurred, and a VM where introspection did occur. In the three environments, we compared the performance cost of virtualization as well as the additional cost of introspection and memory manipulation. Success would mean the introspected VM executed the modified *sensors* program with similar runtimes to the other two environments.

To time our various environments, we created a Python script to perform multiple system calls that execute *sensors*, both unmodified and modified. For each execution, the individual runtime is calculated and stored. After all iterations were run, the statistics were extracted from each environment, and analysis was conducted to calculate each average runtime, standard deviation, and 95% confidence interval for the average. For the purposes of our research, negative timing results (likely a result of out-of-order execution as well as application caching) were discarded to prevent data skew. The results of our testing follow.

5.1. Bare Metal Machine

The bare metal machine utilized the same test hardware and ran the same Ubuntu 16.04.6 LTS distribution as the virtual machines. *Lm-sensors* was installed and executed with no modification to the *sensors* program. Several sensors were detected out-of-the-box and did not require any additional configuration. This setup effectively acted as a baseline for comparison against the VM test environments. As shown in Table 1, we were able to establish a baseline of 2.886ms with most values falling within +/- 0.5ms of that value.

Table 1. Bare metal timing

Bare Metal Machine Timing (9,944 iterations)	
Average Execution Time	2.886ms
Standard Deviation	0.5ms
95% Confidence Interval	+/- 0.00986ms

5.2. Unmodified Virtual Machine

To establish a comparison between an introspected VM and a non-introspected VM, we chose to utilize the Dom0 VM (as opposed to the guest VM) for a few

reasons. First, per the architecture of the Xen Hypervisor, the Dom0 VMM (virtual machine monitor) is simply another virtual machine that has access to the hardware level, so we were able to execute the *sensors* program without any additional configuration, and meaningful values (i.e., real sensor data) was output. Second, our research design ensured that both the hypervisor and guest were running the same kernels and had access to a similar amount of resources, providing a nearly identical environment to the guest virtual machine. As shown in Table 2, we saw that the execution time of *sensors* increased by over 1ms on average (from 2.886 to 4.022ms) resulting in a wider standard deviation. This is likely attributable to the fact that there is some resource sharing and context switches between the VM and the hypervisor taking place.

Table 2. Unmodified guest VM timing

Virtual Machine No Introspection Timing (9,953 iterations)	
Average Execution Time	4.022ms
Standard Deviation	0.73ms
95% Confidence Interval	+/- 0.01425ms

5.3. Introspected Virtual Machine

Finally, our guest machine was tested with DRAKVUF running simultaneously on the hypervisor with **smokescreen** implemented and running as a part of the DRAKVUF instance. The guest VM executed the modified *sensors* program. As shown in Table 3, introspection incurred a relatively large penalty when compared to the non-introspected VM. The average execution time jumped to 8.85ms, with a much wider range of execution times experienced. This was likely due to a combination of processing required by **smokescreen** and resource sharing described earlier with the non-introspected VM. Although introspection incurred a time penalty of approximately 120%, the actual amount of time is still very small, and could likely be attributed to resource sharing among processes or other high-priority processes preempting *sensors* during its execution.

Table 3. Modified guest VM timing

Virtual Machine With Introspection Timing (9,959 iterations)	
Average Execution Time	8.850ms
Standard Deviation	1.493ms
95% Confidence Interval	+/- 0.04816ms

6. Conclusions

In this paper we designed and implemented **smokescreen** as a DRAKVUF plugin meant to potentially mitigate the capability and device artifacts common in modern VMs. Although other solutions may provide increased fidelity [5], we felt that a solution that existed (mostly) outside of the guest machine was an important factor in achieving our view of a high-fidelity hypervisor. Our results indicate that **smokescreen** provides increased fidelity but at the cost of increased execution time. The benefits and limitations of our solution follow.

6.1. External Solution

Utilizing DRAKVUF [16] allowed us to keep the majority of our solution outside of the guest VM. Our implementation showed that it is possible to redirect execution of artifact-leaking binaries in a way that is difficult to detect through analysis of the original binaries. This results from the fact that there are no indicators that they are *not* the original binaries being executed.

By utilizing LibVMI's API, the limitations of VM introspection quickly became apparent. While introspection does provide access to CPU register and memory contents, it does not allow us access directly to devices or files. This forced our implementation to introduce file system artifacts of its own, since we are unable to modify the file contents of the binaries that leak the device/capability artifacts. Similarly, since we are unable to directly manipulate files within the guest VM through introspection, we do not have the ability to modify the system files that would normally contain the raw sensor values. This required us to modify the various binaries rather than executing our calculations outside the VM and injecting the expected data by the unmodified binaries.

Currently, process injection (where the plugin replaces the binary after it is loaded into memory in the guest VM directly from the hypervisor) is not implemented in **smokescreen**, requiring us to introduce our own file system artifacts into the guest VM. However, when process injection *is* fully implemented, **smokescreen** will act as a natural building block after any self-induced artifacts are retrieved from the virtual machine, achieving our goal of mitigating artifacts through minimal-to-no guest VM modification.

6.2. Performance Implications

Determining the overall impact to our system's performance and whether it would be considered a

significant impact to our hypervisor is an important consideration. The cost of our solution's introspection is a **206%** increase in average execution time of *sensors*. In this case, it is unlikely that the increased amount of time required would raise suspicions, as the difference is less than 6ms between the bare metal machine and our introspected VM. However, as the number of VMs increases and the number of different binaries that are required to be trapped also increases, it is possible that the overall system could experience a more noticeable slowdown. This could become a limiting factor in the deployment of our solution.

7. References

[1] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware analysis via hardware virtualization extensions," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, New York, NY, USA, 2008, pp. 51–62, doi: 10.1145/1455770.1455779.

[2] Xu Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario, "Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware," in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, Jun. 2008, pp. 177–186, doi: 10.1109/DSN.2008.4630086.

[3] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Commun. ACM*, vol. 17, no. 7, pp. 412–421, Jul. 1974, doi: 10.1145/361011.361073.

[4] E. G. Mallach, "On the relationship between virtual machines and emulators," in Proceedings of the workshop on virtual computer systems, New York, NY, USA, Mar. 1973, pp. 117–126, doi: 10.1145/800122.803952.

[5] Y. Zhang, F. Xie, Y. Dong, G. Yang, and X. Zhou, "High fidelity virtualization of cyber-physical systems," *Int. J. Model. Simul. Sci. Comput.*, vol. 04, no. 02, p. 1340005, May 2013, doi: 10.1142/S1793962313400059.

[6] E. Bugnion, J. Nieh, and D. Tsafrir, "Hardware and software support for virtualization," *Synth. Lect. Comput. Archit.*, vol. 12, no. 1, pp. 1–206, Feb. 2017, doi: 10.2200/S00754ED1V01Y201701CAC038.

[7] S. Bansal and A. Aiken, "Binary translation using peephole superoptimizers," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, 2008, pp. 177–192, Accessed: Jan. 20, 2020. [Online]. Available: https://www. usenix.org/legacy/events/osdi08/tech/full_papers/bansal/bansal.pdf

[8] M. Carpenter, T. Liston, and E. Skoudis, "Hiding virtualization from attackers and malware," *IEEE Secur. Priv. Mag.*, vol. 5, no. 3, pp. 62–65, May 2007, doi: 10.1109/MSP.2007.63.

[9] T. Liston, E. Skoudis, "On the cutting edge: Thwarting virtual machine detection," presented at SANS at Night, 2006. [Online]. Available: https://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf.

[10] J. Rutkowska, "Red Pill... or how to detect VMM using (almost) one CPU instruction," The Invisible Things, Nov. 2004. [Online]. Available: http://web.archive.org/web/20110726182809/http://invisiblethings.org/papers/redpill.html.

[11] C. Thompson, M. Huntley, and C. Link, "Virtualization detection: New strategies and their effectiveness," Univ. of Minn., Minneapolis, MN, USA, 2010. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.302.7877&rep=rep1&type=pdf.

[12] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin, "Compatibility is not transparency: VMM detection myths and realities" in Proc. of the 11th Work. On HotOS, 2007. [Online]. Available: https://www.usenix.org/legacy/events/hotos07/tech/full_papers/garfinkel/garfinkel_html/index.html.

[13] A. Ortega, "a0rtega/pafish," GitHub. Accessed on Jan. 14, 2020. [Online]. Available: https://github.com/a0rtega/pafish.

[14] T. Klein, "trapkit.de - ScoopyNG." Trapkit, Accessed Jan. 14, 2020. [Online]. Available: http://www.trapkit.de/tools/scoopyng/index.html.

[15] W3Techs, "Usage of web servers broken down by operating systems." Accessed Jun. 15, 2020. [Online]. Available: https://w3techs.com/technologies/cross/web_server/operating_system.

[16] T. K. Lengyel, S. Maresca, B. D. Payne, G. D. Webster, S. Vogl, and A. Kiayias, "Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System," 2014.

[17] GitHub, "libvmi/libvmi: The official home of the LibVMI project is at https://github.com/libvmi/libvmi." Accessed Feb. 04, 2020. [Online]. Available: https://github.com/libvmi/libvmi.

[18] Rekall Forensics, "Rekall Forensics." Accessed Jun. 19, 2020. [Online]. Available: http://www.rekall-forensic.com/.

[19] Intel, "6th Generation Intel® processor families for S-Platforms, datasheet, volume 1 of 2." Accessed: May 09, 2020. [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/332687.