HyperLeech: Stealthy System Virtualization with Minimal Target Impact through DMA-Based Hypervisor Injection

Ralph Palutke
Friedrich-Alexander-Universität Erlangen

Simon Ruderich
Friedrich-Alexander-Universität Erlangen

Matthias Wild
Friedrich-Alexander-Universität Erlangen

Felix C. Freiling
Friedrich-Alexander-Universität Erlangen

Abstract

In the recent past, malware began to incorporate anti-forensic techniques in order to hinder analysts from gaining meaningful results. Consequently, methods that allow the stealthy analysis of a system became increasingly important.

In this paper, we present *HyperLeech*, the first approach which uses DMA to stealthily inject a thin hypervisor into the memory of a target host, transparently shifting its operation into a hardware-accelerated virtual machine. For the code injection, we make use of external PCILeech hardware to enable DMA to the target memory. Combining the advantages of hardware-supported virtualization with the benefits provided by DMA-based code injection, our approach can serve analysts as a stealthy and privileged execution layer that enables powerful live forensics and atomic memory snapshots for already running systems. Our experiments revealed that HyperLeech is sufficient to virtualize multi-core Linux hosts without causing significant impact on a target's processor and memory state during its installation, execution, and removal. Although our approach might be misused for malicious purposes, we conclude that it provides new knowledge to help researchers with the design of stealthy system introspection techniques that focus on preserving a target system's state.

1 Introduction

The ongoing arms race between malware authors and security practitioners lead to increasingly sophisticated approaches on both sides. Recently, malware began to incorporate anti-forensics to evade analysis. Sparks and Butler [58] presented a novel rootkit technique that subverts the memory translation process of the Windows operating system, and exploits *Translation Lookaside Buffer* (TLB) incoherencies to hide malicious memory. Palutke and Freiling [42], as well as Torrey [61], further enhanced this concept by dynamically virtualizing a victim system's view on the physical memory, relying on a kernel extension. Other approaches use *Direct Kernel Object Manipulation* (DKOM), first discussed by Butler [6], to alter important kernel structures, as memory forensics and live analysis often rely on their integrity [5, 22, 59]. In addition, Zhang et al. [68] bypass state-of-art memory acquisition by manipulating the physical address layout on x86 platforms. Besides attacks that target software-based approaches, Rutkowska [53] demonstrated a method to attack *Direct Access Memory* (DMA)-based acquisition by remapping parts of the *Memory Mapped I/O* (MMIO) address space. Zdzichowski et al. [66] listed further approaches in a recent meta study, surveying the landscape of modern anti-forensics. Approaches like these indicate the necessity for novel analysis techniques that are robust against anti-forensics.

To deliver ideal analysis results, an approach must meet two requirements which seemingly contradict each other: First, the *soundness* of a particular analysis method indicates its robustness against anti-forensics, meaning its degree of accuracy based on the actual data of the current target state. Second, a method's *target impact* implies the amount of modifications it introduces to a target's memory and processor state during its installation, operation, and removal. From a forensics point of view, a low target impact is desirable, as it prevents both a potential loss of evidence and the chance for evasive malware to alter its behavior [31]. Running an analysis tool at the same or even a lower privileged domain gives malware the chance to intercept its functionality and falsify results. Consequently, a sound analysis cannot be guaranteed. To keep control over a system's operation, security software steadily migrated to higher privileged layers [32]. In contrast to malware infections, the deployment of privileged analysis software mostly depends on a system's regular loading mechanisms. These have a quite significant impact on the target state and usually require root access, both disadvantageous from a forensics perspective. Furthermore, analysis methods are usually deployed after a system has been infected, which gives malware the chance to tamper with their installation. Hence, analysts began to use increasingly stealthy approaches to conceal the deployment of their methods. Stüttgen and Cohen [60] inject a minimal memory acquisition module into an already existing host kernel module with only a small target impact.

Besides the installation of an analysis method, both its execution and removal, as well as the extraction of results, which often makes use of existing communication channels, alter the target state to an even higher degree. In addition, these communication channels might already be compromised, so that the integrity of the transferred data cannot be guaranteed.

With the rise of anti-forensics, security practitioners started to use DMA from external hardware in order to analyze a system [7, 15, 36, 44]. This allows the transparent access of a system's memory without notably impacting its state, as DMA does not interfere with a processor's operation. Since these devices are often hot pluggable, DMA-based approaches offer a significant advantage when targeting production systems, where down times are often not acceptable. As hot plugging allows a method to be deployed even after the infection of a system, it is especially useful for malware analysis. In addition, DMA usually bypasses authorization checks enforced by the operating system. As a downside, Gruhn and Freiling [21] showed that these approaches suffer from a lack of atomicity, since the target is not suspended during the analysis or acquisition process. Consequently, they cannot produce fully sound analysis results.

Virtualization-based approaches provide the transparent analysis of a system from the more privileged hypervisor layer. The respective target is booted inside a virtualized execution environment (respectively VM), enabling the isolated analysis of the system through *Virtual Machine Introspection* (VMI) [18]. Since investigators are mostly confronted with already infected systems running on bare metal, these cannot be virtualized by conventional technologies like KVM [20] or Xen [4], however. This led analysts to use *on-the-fly virtualization*, initially introduced by Rutkowska [52] and Zovi [69], which installs a thin hypervisor through a kernel driver, and migrates the running system into a hardware-accelerated VM for further analysis [29, 39, 47, 65]. Although on-the-fly virtualization greatly improves the analysis of a system, it falls short in several categories. Loading a kernel driver requires root privileges and has significant impact on the target state. Furthermore, an already infected kernel might subvert the installation process altogether.

In this paper, we present *HyperLeech*, the first approach combining transparent DMA-based code injection and on-the-fly virtualization. In contrast to existing solutions, our approach enables the sound analysis of a target system with negligible impact on its processor and memory state. In detail, we

- are the first to use DMA from an external PCILeech device to stealthily inject a hypervisor into a target's memory, bypassing common access restrictions,
- use Intel's *Virtual Machine Extensions* (VMXs) to virtualize a running target by transparently shifting it into a hardware-accelerated VM, and hide our system by setting up *Extended Page Tables* (EPTs), providing an abstraction of the physical memory,
- devise the process of removing our system without leaving detectable traces,
- implement a prototype that is capable of virtualizing running multi-core Linux hosts without notably impacting the target's processor and memory state,
- evaluate the target impact caused by the injection, execution and removal of our system,
- point out the performance impact caused by the injection of our system, and
- discuss possible mitigation strategies, as our approach might be misused as a powerful rootkit.

The remainder of this paper is outlined as follows: Section 2 provides fundamental background knowledge that is necessary to understand our design concepts. In Section 3, we present an architectural overview of the HyperLeech system, and describe its injection and removal. Section 4 evaluates the impact on both the target's state and performance, and discusses possible mitigation strategies. Section 5 briefly surveys related work and possible use cases. Concluding remarks and future research directions are given in Section 6.

2 Technical Background

For a better understanding of our design choices, we briefly outline important technical fundamentals. Consequently, we introduce the PCILeech framework (Section 2.1), explain the mechanics of hardware-supported virtualization provided by Intel's VT-x (Section 2.2), and shed light on the *Advanced Programmable Interrupt Controller* (APIC) (Section 2.3). Readers familiar with the topics can skip these sections.

2.1 PCILeech

Originally developed by Frisk [15], the PCILeech project is a generic attack framework that allows external devices to use DMA over *Peripheral Component Interconnect Express* (PCIe) to inject code into the physical memory of a target system. Due to PCIe offering hot plug functionality, a variety of PCILeech devices can be attached to a system at runtime. Similarly, such devices can be unplugged at any time without causing significant interruptions. PCILeech supports various hardware configurations which need to be flashed with dedicated firmware. For this work, we made use of the *PCIe Screamer* device [3] which is based on the XC7A35T Xilinx 7 *Field-Programmable Gate Array* (FPGA) providing native 64-bit DMA with access rates about 100 MB/s. Over the *Universal Serial Bus 3* (USB3) interface, the device is connected to an external controller system which is used to

control the PCILeech software. To attach PCIe Screamer to a free PCIe slot of the target host, some systems might require specific adapters due to different form factors of PCIe (e.g., Express Card, mPCIe, Thunderbolt). For HyperLeech, we only made use of PCILeech's native DMA support to inject our custom hypervisor into a running target system. Hence, no additional software needs to be deployed on the target side.

2.2 Intel VT-x

To improve the performance of VMs, modern processors provide hardware-supported virtualization. Intel [24, vol. 3C] introduced several *Virtual Machine Extensions* (VMXs) which expand a processor's instruction set to allow unmodified guests to be executed inside a hardware-accelerated VM. It provides the new processor mode *VMX operation* which is further divided into the execution modes *VMX root* and *VMX non-root*. The former describes a privileged mode that runs a hypervisor (or *Virtual Machine Monitor* (VMM)), used to control the VMs and schedule hardware resources. VMX non-root, on the other hand, serves unmodified *guest* systems as a transparent and restricted execution environment. The processor uses *VMX transitions* to switch between the two operation modes. With the occurrence of certain events (e.g., accesses to specific registers, execution of restricted instructions, or the interaction with emulated devices) in VMX non-root mode, the processor generates a *VM exit* which transfers control to the hypervisor. Subsequently, the hypervisor has the chance to handle the fault and resume the guest. To launch and control a VM, the hypervisor must configure a *Virtual Machine Control Structure* (VMCS) for each core. Besides comprising the entire state of the guest, this central management structure determines the events that are to be intercepted by the hypervisor.

Next to VMX, Intel processors provide *Extended Page Tables* (EPTs) that support the virtualization of a VM's physical memory. When enabled, a second level address translation maps the guest's physical memory to the real memory of the host machine. Similar to the conventional paging structures, EPTs provide several access flags that prohibit unauthorized memory accesses. Breaching these access privileges leads to an *EPT violation* which is intercepted and handled by the hypervisor. This gives the hypervisor the chance to restrict the guest from accessing certain memory regions.

2.3 Intel APIC

With the emergence of *Symmetric Multiprocessing* (SMP) architectures, Intel introduced the APIC system to deliver and control external interrupts. Its architecture consists of two components which communicate over the system bus. The *I/O* APIC routes external interrupts to one or more *Local Advanced Programmable Interrupt Controllers* (LAPICs), each belonging to a particular processor core. The LAPICs receive interrupts not only from the I/O APIC, but also from the processors' interrupt pins and other internal sources, and forward them to their respective cores for specific handling. The LAPIC appears as a memory-mapped device, providing its physical base address through the `IA32_APIC_BASE` *Model Specific Register* (MSR). The kernel initializes this register by parsing the host's *Advanced Configuration and Power Interface* (ACPI) tables during the early boot phase. Over time, Intel introduced several successors which enhanced the design of the APIC. While the xAPIC only brought a few minor changes, the x2APIC appears as the latest iteration which is accessed through certain MSRs instead of MMIO. Both modes are supported by modern processors, and can be switched by specifying a certain bit in the `IA32_APIC_BASE` MSR. This requires the system to be rebooted, however.

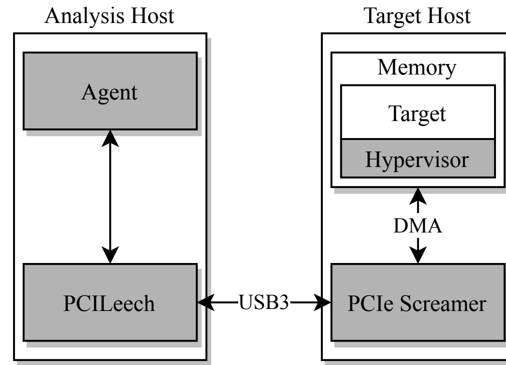

Figure 1: Architectural overview of the HyperLeech system.

Besides interrupts triggered by external devices, the LAPIC provides the possibility to generate *Non-maskable Interrupts* (NMIs). In contrast to maskable interrupts, NMI delivery cannot be trivially deactivated. Each LAPIC provides several *Local Vector Table* (LVT) registers that are used to configure the delivery of different NMI types. Among others, these include timer interrupts, thermal sensor interrupts, and performance counter overflows. The *mask bit* in an LVT allows to disable the delivery of the corresponding NMI type by preventing the LAPIC from forwarding the interrupt to its processor. Once set, further incoming NMIs of the same type set the *pending bit* in the same LVT to signal an outstanding interrupt. The pending NMI is not delivered to the processor until the mask bit in the respective LVT has been cleared. While being disabled only one upcoming NMI can be kept pending. Any additional NMI is lost.

3 System Overview

This section provides an architectural overview of the HyperLeech system, and illustrates its injection (Section 3.2) and removal (Section 3.3) mechanisms. The basic architecture

of our prototype comprises several components which are grayly depicted in Figure 1. HyperLeech targets a physical host which is subject to be analyzed. As the target will exclusively be accessed from external hardware, no login credentials are required to inject our *hypervisor* into its memory. To access the memory of the target host, we flash the PCILeech firmware to the *PCIe Screamer* FPGA, and attach it to a free PCIe slot on the target side. PCIe Screamer allows native 64-bit DMA operations, and thus access to the target's entire physical memory. Over USB3, PCIe Screamer is connected to an analysis machine that is used to execute the controlling *agent* software. The agent, written in Go, serves the analyst as an interface for controlling our hypervisor through the *PCILeech* host software.

3.1 Mode of Operation

For the installation and removal of our hypervisor, the agent uses DMA to inject multiple code stages into the target memory. This is possible as x86-64 ensures cache coherency regarding DMA operations, preventing processors from retrieving inconsistent data [24, chap. 11.3.2]. The stages are designed to preserve memory and processor state, so that the target is not notably impacted by the injection (see Section 4.1). Apart from this, we outsource most computational tasks to the remote agent in order to further reduce target modifications.

Prior to the code injection, the agent needs to determine the location of the target kernel. Due to the usage of *Kernel Address Space Randomization* (KASLR), modern Linux kernels are randomly placed in physical memory. Therefore, the agent scans the entire physical memory until it matches a pre-registered signature of the kernel's first code page. To correctly terminate the scanning process, the agent issues DMA read operations to probe the amount of memory installed in the target host. As certain memory ranges do not respond to DMA (e.g., MMIO areas which are used to map certain devices), a timeout is employed on each read operation to prevent the agent from stalling.

Once the kernel is found, the agent writes special *injection stages* (see Section 3.2) into the target's physical memory. These stages hijack the control flow of the target kernel, and subsequently install a thin hypervisor that virtualizes the system at runtime. Despite being able to write arbitrary memory over DMA, taking over the system's operation is mandatory to actually execute the injected code. For a stealthy operation, we designed a lightweight, VMX-based custom hypervisor that mostly avoids any interference with the target's operation. Limited to DMA, the agent cannot obtain contextual information of the running system, as it is restricted to a physical view. Therefore, the agent risks potentially corrupting the target by overwriting the memory which is currently in use. To minimize this risk, the agent searches for regions that are unlikely to be used. During our research, we determined the first two KiBs of a Linux kernel's code segment to be the perfect injection spot as it mostly consists of `nop` instructions which do not have any functional purpose. As of that, modifying this *nop area* does not corrupt the kernel's execution. Prior to overwriting any memory, the agent stores the original content to the analysis machine, so it can be restored in due time. Since the nop area does not hold enough space for the actual hypervisor, the stages request the target kernel to allocate further memory. This is not an issue from a forensics perspective, as occupying currently unused memory should not corrupt evidential data in most cases. After receiving the necessary information, the agent sets up the hypervisor's memory layout, and copies the appropriate page tables and the binary of the hypervisor to the previously allocated memory. Withdrawing control from the target leads the hypervisor to take over and virtualize the running cores. At this point, the hypervisor is in full control, ready to fulfill stealthy analysis tasks or perform memory forensics. Eventually, the agent removes the injection stages, restoring the original target memory.

Using DMA, an analyst can now instruct the agent to transparently interact with the hypervisor, allowing to send commands or receive data. Compared to conventional communication channels that rely on the file system or a network card, exchanging data over DMA is both stealthier and less intrusive to the target state. Especially for targets that might be compromised, covert communication is an important requirement for the integrity of the transferred data.

To deinstall our system, the agent overwrites parts of the target's kernel space with *removal stages* (see Section 3.3) which have the purpose of transparently devirtualizing the system. After that, the hypervisor signals the agent to restore the overwritten memory including the hypervisor area. At this point, the target is resumed without leaving any traces of our system.

3.2 Injection

We designed the injection process in multiple *injection stages*, so that no relevant data is corrupted during the virtualization of the target system. We minimized dependencies on the target kernel, as the system could have already been compromised, and thus subvert the injection process. One exception is to query the target kernel to allocate some memory for the *hypervisor area*. This seems acceptable, however, as we doubt that malware could forge the allocation without risking severe system failures. Figure 2 gives an overview of the injection. While stage I1 only consists of a few bytes overwriting a kernel function to hijack the control flow, I2 and I3 are written to the previously mentioned nop area. In contrast, I4 and I5 are directly placed within the hypervisor area. Upcoming, we shed light on the individual injection stages.

Stage I1: Control Flow Hijacking Once the target kernel's base address was found, an injection location which allows to take over a processor's execution is chosen. In contrast

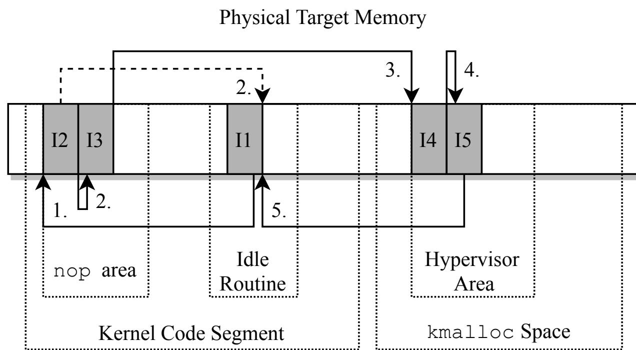

Figure 2: View of physical memory after the injection stages I1 to I5 were written to the target memory. The hypervisor area was vicariously allocated by the target kernel. Arrows represent jumps to subsequent stages. The dashed arrow symbolizes the abort of the injection, jumping back to the kernel's idle routine.

to the base of the kernel which differs after each boot due to KASLR, offsets within its code segment only depend on the actual kernel version. This is because the Linux kernel is linearly mapped in both physical and virtual memory. Consequently, addresses of arbitrary kernel symbols can be statically computed for a particular target by adding the respective offsets to the previously determined kernel base address.

To actually hijack the target kernel, we chose to hook its idle routine (`intel_idle`), as it is regularly executed by each processor and allows to run code with ring 0 privileges which are required for the virtualization. Other valid injection spots include a system's scheduler as well as its interrupt handlers. For the later, special care has to be taken since the injected code needs to run in interrupt context, however. To decrease the latency a processor takes to enter our hook, even multiple injection locations could be selected. As long as the hook is placed within the target's kernel space and is regularly executed, almost any location could be used to hijack the control flow. This prevents the target system from mitigating our injection by monitoring specific memory regions.

As soon as a core starts idling, `intel_idle` is invoked, entering a sleeping state to save power until it is woken up again. Before placing the hook, the agent saves the respective memory to be able to restore it later on. Acquiring memory over non-atomic protocols like DMA introduces a race condition, as the kernel might alter the area before it enters the hook. As the kernel's code segment should be mapped non-writable at all times, it usually is never modified, however. Nevertheless, a custom kernel might remap its code ranges to be writable. Even mainline kernels might sometimes alter parts of their code segments when using instrumentation frameworks like Ftrace. When activated, Ftrace dynamically modifies the first few bytes of a kernel function, so that subsequent invocations detour to a custom handler before returning to the originally intended code. Therefore, we avoided to overwrite these first few bytes with our hook. Instead, the agent replaces the following bytes with a relative jump to stage I2. As we could have chosen any other spot in the kernel code, monitoring the idle loop is not a reliable method to generically detect our approach. To virtualize every core, the hook must remain until all processors have passed the residual stages. Our current implementation relies on target kernel functionality to determine the number of target cores (`num_online_cpus`). However, enumerating the *Non-Uniform Memory Access* (NUMA) hierarchy could possibly provide a more target independent way. After entering the hook, the core jumps to stage I2.

Stage I2: Processor Serialization Until having allocated further memory for the hypervisor, we use the kernel's stack to temporarily store register content that is about to be modified, preventing a loss of processor state. As discussed in Section 4.1, this appears to be insignificant from a forensics perspective. The subsequent process is sequentialized by the possession of a global lock which prevents the target cores from concurrently entering the following stages. In case the lock has already been occupied, a core immediately detours to a specifically prepared trampoline that is responsible for executing the instructions overwritten by our hook, before resuming the original execution within the idle loop (dashed arrow in Figure 2). Thus, no processor stalls while the lock is held by another core. The trampoline mechanism is also used for processors that were already virtualized, as these must be prevented from reentering the subsequent stages. To determine the current processor's virtualization state, I2 attempts to force a VM exit which is only generated in VMX non-root mode. In case of an already virtualized core, this forces a context switch to the hypervisor which in turn informs I2 about the current processor's operation mode. Whenever a non-virtualized core acquires the lock, it is allowed to enter the subsequent stages, eventually leading to its virtualization.

Stage I3: Hypervisor Setup To prevent the target from disrupting the installation of our hypervisor, I3 temporarily disables all interrupts until the processor reaches the end of I4. Besides maskable interrupts, modern processors support NMIs for critical asynchronous event delivery like hardware interrupts or watchdogs. These special kind of interrupts cannot be trivially disabled with the `cli` instruction, however. To temporarily deactivate NMI delivery, I3 reconfigures each processor's LAPIC, disabling the valid flags of its LVT registers (see Section 2.3). Additionally, it consults the `IA32_APIC_BASE` MSR to determine the system's current APIC mode, as both xAPIC and x2APIC are supported by modern processors. Consequently, our implementation offers different ways to access a LAPIC. As NMIs must be disabled before switching to the hypervisor's memory layout, there

is no way to map the LAPICs' physical addresses. Therefore, we decided to rely on the kernel's `APIC_BASE` symbol to make use of the already established kernel mapping. From a forensics view, we do not expect this symbol to be a critical target dependency, as it is defined as a kernel constant.

For the memory of the actual hypervisor, I3 manually calls the kernel's `kmalloc` function to allocate additional space. Our experiments revealed that a single two MiB page seems to be a sufficient size. Most of the hypervisor area serves for dynamic memory allocations during the setup of the hypervisor. Other parts are used to store its code, data, stacks, and page tables. The resulting base address of the hypervisor area is then translated to its physical counterpart, and provided to the agent using DMA. After receiving the information, the agent copies relevant parts of the hypervisor to the newly allocated area. These include a custom memory layout which exclusively maps the hypervisor and the injection stages. Depending on the specific use case, mapping certain parts of the guest's address space might be conceivable. Eventually, the processor's TLBs are flushed, the newly created memory layout is enabled, and *Process Context Identifiers*s (PCIDs) are deactivated to prevent caching leftovers.

Stage I4: NMI Handling Entering stage I4 first sets up a processor individual stack which is subsequently used to store the state of the core. To prevent stack overflows from corrupting any memory, the stacks are surrounded by non-presently mapped pages.

Before NMIs can be safely reenabled, I4 registers a custom handler which ensures NMIs that would otherwise be lost to be reinjected into the guest. This additionally requires the installation of both a custom *Global Descriptor Table* (GDT) and *Interrupt Descriptor Table* (IDT). During the execution of the hypervisor, the handler records upcoming NMIs within a bitmap which indicates if a processor has an NMI pending. To distinguish the running cores, and thus mark the correct bit within the pending bitmap, the hypervisor's NMI handler compares the stack pointer of the current processor with the hypervisor's stack ranges. As every virtualized core has its own stack which is known to the hypervisor, these information can be used to distinguish the processors without the need to consult the target kernel. Until properly acknowledged by clearing the pending flag in the respective LVT, an NMI is not forwarded to the corresponding processor core. This, however, prevents further NMIs of the same type from being delivered independent of the valid flag of the corresponding LVT register. As a solution, the hypervisor consults the pending bitmap to determine if an NMI must be reinjected to the current virtualized core every time it is about to reenter the guest. This is necessary, as watchdog tasks could potentially misbehave due to missing NMIs, and thus corrupt the target state. To inject an NMI into the guest, the hypervisor sets the *valid flag* of the *VM-entry interruption-information field* within the corresponding VMCS, and specifies the *interruption type* field to indicate an NMI. This automatically invokes the guest kernel's NMI handler as soon as the guest is resumed. From the perspective of the guest, it is not distinguishable if an NMI appeared during its own operation or due to an injection from our hypervisor. The guest's NMI handler then clears the pending flag in the respective LVT, allowing NMIs of the same type to be delivered again. Lastly, I4 reenables all interrupts, and jumps to the final stage.

Stage I5: Processor Virtualization To preserve the full register state of a processor, all previous stages had to be implemented in plain assembly code. With a custom stack being set up during the previous stage, I5 is implemented in the high-level programming language C. The stage is responsible for the setup of our hypervisor and the actual virtualization of a processor. Consequently, it allocates relevant data structures like the VMCS within the hypervisor area. With all the preparations done, the processor releases the global lock before entering the virtualization process. Provided *Intel Virtualization Technology* (VT-x) was not deactivated in the *Unified Extensible Firmware Interface* (UEFI) or BIOS settings (which is not the case in most modern machines), I5 enables VMX operation, and copies the previously saved processor state to the VMCS. Similar to a hypervisor rootkit [42, 52, 69], this allows to transparently launch the target system inside a hardware-assisted VM, and resume its original execution with its current state. To remain stealthy, we configure the VMCS so that the hypervisor intercepts only a minimal set of events. Thus, hardware is directly passed to the guest without the hypervisor's interference. This increases the overall performance of the guest while reducing detectable side channels. Depending on the specific use case, it might be necessary to configure the interception of additional events, however. Due to the hypervisor area being allocated by the target kernel, it should never be accessed by accident. To prevent the target from purposely accessing the hypervisor area, we set up EPTs to redirect read accesses to a 4 KiB *guard page*. Write accesses are configured to generate an EPT violation in order to keep the originally stored memory on the analysis machine up to date. Consequently, the hypervisor intercepts the write accesses, and notifies the agent to update its copy. The remaining physical address space is identity mapped. Eventually, the hypervisor resumes the target's original execution inside the idle routine, now running as a virtualized guest. As we chose a symmetric hypervisor design, the entire injection process needs to be repeated for each core. From there on, the hypervisor can be used for any specific task while stealthily controlling the target system.

3.3 Removal

Similar to the injection, the removal of the hypervisor requires the agent to copy several *removal stages* to the target memory. These stages devirtualize the target, and clean up its memory

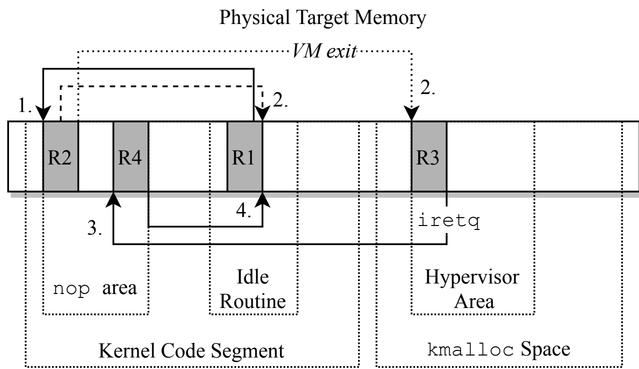

Figure 3: View of physical memory after the removal stages R1 to R4 were written to the target memory. The arrows represent jumps to subsequent stages. The dashed arrow symbolizes the abort of the removal process, returning to the kernel's idle routine. The dotted arrow depicts a mode switch to the hypervisor running in VMX root mode.

and processor state. Rebooting the system will also remove the hypervisor, as it only resides in memory and is currently not configured to intercept and emulate system shutdowns. This is not a conceptual limitation, however. Figure 3 illustrates the injected removal stages within the physical memory of the target host. Once again, the first stage R1 appears as a hook within a permanently invoked kernel function, enabling HyperLeech to take over control. R2 and R4 are placed within the nop area, and R3 is part of the hypervisor's code base. The remains of this section provide brief information about the individual removal stages.

Stage R1: Control Flow Hijacking To devirtualize the target system, a context switch to VMX root mode is mandatory. As we cannot rely on the guest to trigger a VM exit, we decided to reinstall a hook within the idle routine (`intel_idle`) to withdraw control from the target kernel. Like with the injection process, entering the hook transfers each processor to the subsequent stages. Once again, care had to be taken to inject the following stages before installing the hook.

Stage R2: Processor Serialization Similarly to the injection process, the trampoline mechanism prevents certain cores from entering the subsequent stages (dashed arrow in Figure 3). This time, however, already devirtualized processors detour to our trampoline, execute the overwritten instructions, and return to the idle loop. In addition, a global lock once again ensures the serialization of the following stages. Failing to acquire the global lock also results in the immediate return to the idle routine after detouring to the trampoline. Here on after, R2 forces another context switch to the hypervisor which provides the next stage (dotted arrow in Figure 3).

Stage R3: Processor Devirtualization Stage R3, now being executed in VMX root mode, first determines if the guest's current instruction pointer refers to the code of R2. This verifies that the context switch was indeed caused by R2 indicating a valid unload process. Although standard kernels would not map any legitimate code to their nop area, a custom kernel might deviate from this behavior, forcing our hypervisor to be unloaded. Therefore, the agent is required to approve the removal process beforehand. Otherwise, the hypervisor simply ignores the unload request, even if it came from the correct address range. In case of a legitimate removal, stage R3 disables interrupt delivery. NMIs that occurred up to this point would be lost after the hypervisor's deinstallation, as these could not be reinjected into the guest anymore. This, however, would block further NMIs from being delivered, as the target kernel would not be aware of the pending interrupt. To avoid this issue, our hypervisor aborts the devirtualization in case of a pending NMI, reinjects it, and waits for the target to reenter the idle loop, restarting the removal process. As this time window is relatively small, NMIs barely came across during our experiments, however. After disabling interrupts, VMX operation can safely be terminated, effectively devirtualizing the core. Then, R3 restores the suspended guest processor state by consulting the respective VMCS. Via the `iretq` instruction, the remaining processor state is restored and control detours to stage R4.

Stage R4: Hypervisor Cleanup Once a processor was devirtualized, stage R4 restores the memory layout of the target and releases the global lock, allowing further cores to proceed with the removal. Afterwards, interrupts are reenabled, as these can now be handled by the target. To conceal any traces, the last core entering R4 signals the agent to restore the hypervisor area. As the saved copy was constantly updated during the interception of write accesses, it always contains the current state. It is then freed by issuing the kernel's `kfree` function. From there on, the target has no chance to detect any evidence of the previously existing hypervisor. Subsequently, R4 jumps back into the idle routine, resuming the original execution of the target. After all cores have been devirtualized, the agent restores the memory of the injected removal stages, preventing traces from being left over.

4 Discussion

We tested our implementation on a target host running an Intel Sandy Bridge i5-2400 processor supporting VT-x, with four GiB of memory, and a Fujitsu D3062-A1 motherboard. We installed Debian Stretch with the Linux kernel version 4.9.88-1.deb9u1 as its operating system. Since HyperLeech is mostly operating system agnostic, only minor adaptions are required to support newer target kernels. Inserting the PCIe Screamer card [3] which is flashed with the PCILeech firmware version

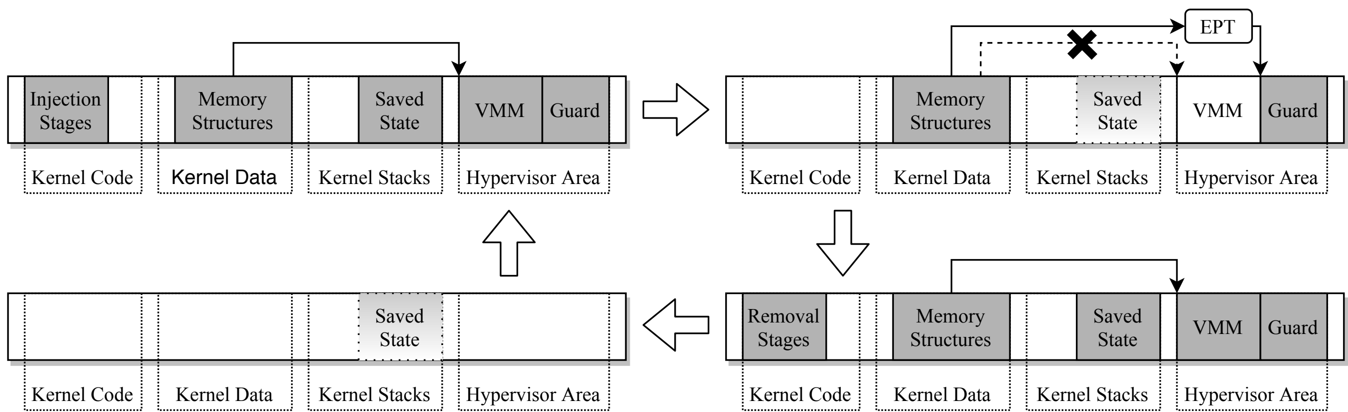

Figure 4: Impact on the target memory during the injection (top left), execution (top right), and removal (bottom right) of the HyperLeech system. Bottom left depicts the memory state after our system was fully removed. *Memory structures* represent kernel structures that reference the hypervisor area due to the allocation via `kmalloc`. During the execution, these references are redirected to the hypervisor's *guard page*. *Saved state* depicts processor registers that are temporarily saved on the kernel stacks.

3.5 [17], grants DMA to the target's memory. On the analysis machine, we used Windows 10 Enterprise (revision 1709) which required the installation of the FTDI USB drivers [1] to communicate with the PCILeech firmware.

4.1 Target Impact

This section discusses the modifications of the target's processor and memory state, which arise due to the injection, execution, and removal of the HyperLeech system. Our main design consideration was to reduce the impact on the target while minimizing dependencies on kernel functionality.

Injection During the installation and removal of our system, the agent replaces certain parts of the target memory with the *injection stages* (Figure 4, top left). To preserve the original memory, the agent saves the respective areas for later restoration. Executing injected code within the target kernel typically leads to further modifications. Thus, each stage is designed to best preserve the target's memory and processor state. As multiple cores could enter the first two injection stages in parallel, we use the kernel stacks to store processor state (called *saved state* in Figure 4) that is about to be modified. This also simplifies our implementation, as it renders error-prone synchronization mechanisms obsolete. Pushing state onto the kernel stacks cannot corrupt data that is still required by the target, as even red-zones are disabled for proper interrupt handling. Upon entering stage I3, the remaining injection is serialized by a global lock. Consequently, we store further data directly within the memory of stage I3 (and thus in the kernel's `nop` page), as it is restored anyway and does not require any synchronization. With I4 being entered, custom stacks for each processor are allocated within the hypervisor area. These are subsequently used to store a processor's state.

Execution After HyperLeech has been installed (Figure 4, top right), the agent restores the injection stages with their original content. This leaves only minor modifications of the *kernel stacks* and a few *memory structures* which reference the hypervisor area due to the allocation via `kmalloc`. As the configuration of EPTs redirects all read accesses targeting the hypervisor area to a *guard page*, following these references won't find any suspicious traces. The target kernel assumes this area to be legitimately in use anyway, and usually won't ever access it until it is freed again. Write operations that target the hypervisor area are intercepted, and the originally stored copy is updated on the analysis machine. We consider the modifications of the kernel stacks to be practically undetectable, as these are almost instantly overwritten by regular kernel usage once the target is resumed.

Removal Removing our hypervisor requires the injection of the *removal stages* which are repeatedly used to devirtualize the system (Figure 4, bottom right). As with the injection, R1 hijacks the target's control flow, while R2 serializes the remaining process. Once again, the kernel stacks are used to preserve target processor state that is about to be modified in the meantime. R3 devirtualizes a core and restores the current guest processor state. The final core entering stage R4 frees the hypervisor memory after its has been restored by the agent. As the stored copy was constantly updated during the hypervisor's operation, it always contains the current content. Here on after, the hypervisor area is queued back into the kernel's slab allocator, effectively deleting the referencing *memory structures*. As a last step, the *removal stages* are overwritten by the agent, restoring the original target memory (Figure 4, bottom left). With the exception of the negligible modifications of the kernel stacks, no processor or memory state is ever

lost during the injection, execution, or removal of our system. Moreover, the overwritten parts of the kernel stacks do not contain any relevant data, and the modifications are almost instantly overwritten as soon as the target resumes its execution. Especially for the purpose of memory forensics, where evidential integrity plays an important factor, HyperLeech seems to be a promising step in the right direction.

PCIe Screamer Although the injection, operation, and removal seemingly have no notable target impact, attaching the PCIe Screamer card introduces detectable modifications. This is because the PCIe bus is enumerated whenever the kernel registers a new device. The enumeration introduces changes to the file system and leads to modifications of the memory and processor state. For the target, however, PCIe Screamer only appears as a Xilinx ethernet adapter which could further be adapted by altering its device id. Thus, the target cannot refer the device to our system, as it is not distinguishable from any legitimately added hardware. Section 6 presents an alternative injection method that avoids the necessity of attaching hardware altogether, so that even changes caused by the enumeration could be prevented.

4.2 Performance Impact

This section summarizes the performance impact of the injection. In this course, we measure the duration of the virtualization of each target core (*Core X*), and compare the cumulative *sum* to the time required by the *full* injection process. For better results, we repeated the measurements for five iterations after resetting the target each time. Table 1 summarizes our results. Comparing the measurements of the individual processors, the first core takes three times longer to finish the injection. This is because the first core is responsible for executing additional tasks, e.g., requesting the target to allocate the hypervisor area, waiting for the agent to establish a custom memory layout, and copying the hypervisor to the target memory. Compared to the cumulative *sum* of the measured durations, finishing the *full* target virtualization lasted significantly longer. This is due to preparation and cleanup steps of the agent, as well as the virtualization being serialized by a global lock during the second stage (see Section 3.2). Consequently, only one core at a time is able to progress through the remaining stages. While the lock is occupied, all other processors resume the target's original execution until they retry the virtualization when entering the idle routine the next time. Although the serial approach leads the entire injection to last longer, no processor has to stall its original work.

As we designed our hypervisor to intercept only a minimal set of events, its performance impact during its execution appears to be minimal [2]. Depending on the actual use case, this overhead might change, however.

Table 1: The time each *core* takes to be virtualized, measured over five different runs. While *Sum* cumulates the durations of each processor run, *Full* informs about the total duration of the entire target virtualization. The bottom row visualizes the mean values of the individual runs.

Core 0	Core 1	Core 2	Core 3	Sum	Full
225 ms	66 ms	64 ms	71 ms	426 ms	6784 ms
233 ms	66 ms	63 ms	71 ms	433 ms	1652 ms
545 ms	76 ms	76 ms	65 ms	762 ms	1561 ms
597 ms	72 ms	87 ms	64 ms	821 ms	4029 ms
249 ms	71 ms	65 ms	64 ms	449 ms	1627 ms
369 ms	70 ms	71 ms	67 ms	578 ms	3132 ms

4.3 Memory Acquisition

To counter anti-forensics, analysts often acquire a system's volatile memory for subsequent analysis [32]. This has the advantage that malware cannot actively hide once the snapshot has been acquired. Vömel and Freiling [62] introduced the three criteria *correctness*, *atomicity*, and *integrity* to assess the quality of an acquisition method. *Correctness* determines the differences between the dump and the actual memory content acquired at a certain time. Thus, malware that tampers with the acquisition process could impair the correctness of a dump [5, 22, 68]. To consider a memory dump as *atomic*, the acquisition process must not be affected by the target system's concurrent activity. Since memory is mostly acquired at a system's runtime, a correct memory image does not inevitably imply atomicity. Lastly, the criterion *integrity* measures to what extent memory content is altered by the acquisition method itself. As most acquisition software directly runs on the target host, certain memory needs to be overwritten by its own code and data. These criteria can be mapped to the analysis requirements *soundness* and *target impact*, defined in Section 1. While a sound analysis method must inherently be correct and atomic, the target impact can be seen as the pendant to the criterion integrity.

Various approaches emphasized the benefits of acquiring volatile memory through on-the-fly virtualization [29, 39, 65]. Most rely on a copy-on-write mechanism that provides both full atomicity and correctness for the acquired image. However, full integrity cannot be achieved, as the installation of these systems has a substantial impact on the particular target state. Other approaches perform DMA to acquire memory without having a significant impact on the target state [7, 36]. However, as DMA does not interfere with a processor's execution, atomicity cannot be guaranteed.

HyperLeech was specifically designed to minimize the target impact, and thus advances memory acquisition to achieve much better integrity compared to existing approaches. The previously mentioned copy-on-write mechanism could be integrated into our system. While it would also achieve full atomicity and correctness, deploying our system to the target

host would require severely less state modifications (see Section 4.1). Section 6 discusses further enhancements that might allow memory acquisition to even fully satisfy all three criteria. Further conceivable use cases are outlined in Section 5.

4.4 Mitigation

As our system could also be misused as a malicious hypervisor rootkit [42, 52, 69], we discuss approaches that either prevent or at least detect its presence.

Prevention As HyperLeech relies on hardware support to set up a VM, the virtualization of the target cores can be mitigated by disabling VT-x in the UEFI or BIOS respectively. This, however, would also deprive the possibility to run legitimate VMs, and is thus often not a valid option. Alternatively, a target system might entirely prohibit the attachment of new PCIe devices, preventing our agent from accessing the target memory. However, this would also hinder a user from legitimately connecting additional PCIe hardware.

A better solution might be the restriction of DMA from untrusted sources. Modern systems provide an *Input/Output Memory Management Unit* (IOMMU) which functions similar to a standard *Memory Management Unit* (MMU), as it traverses certain mapping tables which specify memory access permissions. In contrast to regular memory accesses, an IOMMU controls DMA issued by devices, however. Hence, the target could configure the IOMMU to protect its kernel memory from illegitimate accesses. Despite an IOMMU being the best way to prevent DMA-based injections, most modern systems still do not enable it by default. This includes not only recent Linux distributions, but also Microsoft's latest operating system Windows 10 [8]. Only Apple enforces the activation and usage of an IOMMU since MacOS High Sierra [16]. However, even in case an IOMMU is in use, it has to be properly configured by the kernel and firmware respectively. Recently, Markettos et al. [37] elaborated how PCIe features like *Address Translation Services* (ATS) might enable a malicious device to act benignly, effectively bypassing an even properly configured IOMMU. According to Morgan et al. [40], a large part of today's machines grants DMA to all peripherals prior to the configuration of the IOMMU during the boot sequence for compatibility reasons. Furthermore, Markuze et al. [38] discuss the possibility to exploit trusted devices, as these are often not restricted by the IOMMU. These publications show that even the usage of a correctly configured IOMMU might not be able to prevent our approach.

As a target's processor, chipset, and mainboard must support the IOMMU in the first place, this might not even be a possibility for older machines. Instead, a system might clear the *bus master enable* flag to disable DMA for specific devices or upstream bridges. Nevertheless, Windows 10 seems to be the only system which, by default, uses this mechanism to prevent hot-pluggable devices from using DMA during screen locks, as stated by Delaunay [8].

Detection Since the entire installation only requires a few seconds (see Section 4.2), and no significant target state is altered, detecting our system during the injection phase seems unlikely. As soon as the target has been fully virtualized, meaningful changes made to the kernel are reverted (see Section 4.1). Here on after, detecting HyperLeech is practically limited to finding indications of the actual virtualization. Intel [24, vol. 3C] designed its virtualization extensions to be transparent to a guest system. Nevertheless, researchers suggested different kinds of side channels to detect a VM. In addition, several researchers discussed the possibility to exploit timing discrepancies to find out whether a system is running inside a VM [19, 48, 54, 55]. As virtualization adds the additional hypervisor layer beneath the already running system, certain events must be intercepted to stay in control over the VM. Both the context switches and the actual handling of these events introduce a runtime overhead which can, theoretically at least, be detected. As VMX provides the possibility to forge internal clocks of the guest, this overhead can be concealed, however. Moreover, George [19] mentioned that a guest could rely on external timers, but these are often inaccurate and require the usage of additional protection to prevent them from being forged. Since virtualization becomes more prevalent, and HyperLeech could possibly attack already virtualized targets via nested virtualization (although currently not being implemented), relying on the detection of virtualization might not be sufficient in the future anyway.

5 Related Work

Initially presented with Tribble [7] and Copilot [44], various projects used DMA to access a system's memory over protocols like PCI, PCIe, or FireWire [67]. However, several researchers showed that DMA-based approaches suffer from concurrency issues and a lack of context information about a target's execution [21, 25]. This prohibits security related tasks like tracing, debugging, and control flow analysis. Frisk [15] extends common approaches, injecting kernel implants which execute custom code within a target's kernel space. Duflot et al. [11] exploited vulnerable firmware of a network card, using its DMA capabilities to take over the target host. Both, however, significantly modify the state of the target system. In contrast, HyperLeech is optimized to preserve a system's state and operate transparently with only a minimal impact on the analyzed target.

King et al. [28] were the first to use software emulation to stealthily virtualize a victim system. However, with the advent of hardware-supported virtualization, rootkits were able to shift already running systems into VMs without requiring a reboot. While Rutkowska [52] targeted Windows

Vista kernels, Zovi [69] attacked MacOS systems. Recently, Palutke and Freiling [42] adapted this approach attacking Linux systems, and enhanced it by locating their rootkit in *hidden memory*, certain address ranges that are not even visible to the operating system's kernel. Hypervisors have not only been used for offensive purposes, however. Approaches like [29, 39, 49, 65] perform live forensics, atomically extracting information from a guest's memory via VMI [18]. While some of these approaches require the target system to be booted inside a VM, others virtualize a system at runtime [46]. Further projects instrument a guest by injecting breakpoints into its memory, and hide these modifications by exploiting TLB incoherencies [9, 34, 47]. Fattori et al. [13] introduced an on-the-fly hypervisor that serves analysts as debugger for guest operating systems. Moreover, several approaches perform stealthy system tracing [10, 45, 51, 57, 63]. Korkin and Tanda [30] present methods to transparently control memory accesses from a hypevisor. Nguyen et al. [41] analyze malware, using a lightweight virtualized execution environment, while Rhee et al. [50] rely on a hypervisor to protect kernel structures from rootkits. Sharif et al. [56] bridged the semantic gap between the hypervisor and its guest by transparently performing malware analysis from inside the guest. In contrast, Jiang et al. [26] shift the analysis techniques outside of the guest, overcoming the semantic gap by systematically reconstructing internal semantic views. Yan et al. [64] combined hardware virtualization and software emulation to comprehend malware actions. To reveal malicious processes hidden by rootkits, both Jones et al. [27] and Litty et al. [35] suggested hypervisor-based methods to detect hidden processes by reconstructing guest information. While HyperLeech currently serves as a stealthy execution layer only, all of the above mentioned approaches might be adapted to our system. Existing approaches either need to be deployed via kernel drivers, inferring significant changes to the target system, or require the target to be booted inside a VM. While loading a kernel driver implies root privileges and support from the target kernel, booting the target inside a VM excludes systems that already run on bare metal. Besides, custom kernels might prohibit to load kernel drivers altogether. HyperLeech stealthily injects a hypervisor through DMA, allowing the virtualization of a target without the necessity to deploy a driver or to posses root privileges. In addition, the installation of our system cannot be easily inhibited by malware or intrusion prevention systems that monitor the loading of new kernel components. This is especially useful for forensic approaches, as these often need to be deployed after a system has potentially been compromised.

6 Conclusion and Future Work

To counter sophisticated anti-forensic approaches, the transparent analysis of a potentially compromised system became increasingly important. With this paper, we presented a novel method which uses DMA provided by a PCILeech device to inject a hypervisor into a running system's volatile memory without requiring access privileges. With negligible impact on processor and memory state, HyperLeech is capable of transparently virtualizing modern multi-core Linux hosts, serving analysts as a stealthy and privileged execution layer. Compared to our approach, others rely either on virtualization based on the loading of a kernel extension, causing severely more state modifications, or suffer from a loss of context information and atomicity, being restricted to DMA. As most of today's systems do not offer appropriate protection against DMA from external devices, we expect HyperLeech to be functional on a wide variety of machines. In conclusion, our approach advances modern system analysis and memory forensics, enabling investigators to achieve sound results even in compromised environments. In the following, we point out further research directions for enhancing our current system.

Due to the configuration of EPTs, our hypervisor isolates itself from the target guest. However, EPTs can only restrict conventional memory accesses issued by the memory controller, and can be bypassed via DMA. As our hypervisor is placed within a memory region that was vicariously allocated by the target kernel, the guest should never access this area by accident. However, the target might intentionally issue DMA operations to scan its own memory for conspicuous traces. To protect the hypervisor from DMA, HyperLeech must properly configure an IOMMU, using Intel's *Virtualization Technology for Directed I/O* (VT-d). This way, the hypervisor would be fully protected from both conventional memory accesses (via EPTs) and DMA (via the IOMMU).

Attaching the PCIe Screamer device to a target host introduces a notable impact on the target state (see Section 4.1). To avoid unintended modifications, already existing management co-processors like Intel's *Management Engine* (ME) [23] or a *Baseboard Management Controller* (BMC) could be used instead. These co-processors are typically used to execute software that controls and monitors the actual host. Although mostly being signed and protected, researchers showed various ways to deploy custom modified code to run on such platforms [12, 43]. Furthermore, the open-source implementation OpenBMC could be adapted to run custom code on a BMC without requiring to exploit a vulnerability [14]. Usually, these co-processors provide their own DMA engines enabling access to the host memory for efficient data exchanges. Recently, Latzo et al. [33] presented a patch for OpenBMC running on ASPEED's AST2500 *System-on-Chip* (SoC), using it as a PCILeech device. From the host's perspective, the SoC appears as an arbitrary graphics card that connects over PCIe. This could allow the injection of the HyperLeech system without the necessity to attach additional hardware. Therefore, this would prevent the target system from detecting modifications caused by the PCIe enumeration. As a result, analysts could acquire data in a completely sound way while seemingly having no impact on the target at all.

Furthermore, virtualized targets should be able to launch their own VMs. This requires our hypervisor to provide nested virtualization, as VMX allows only one hypervisor to run in VMX root mode. Even in case the target already runs a hypervisor, our injection could be adapted to take over by withdrawing control from certain routines that are repeatedly executed in VMX root mode. Consequently, even virtualization-based rootkits could not prevent our system from being deployed unless they correctly configure an IOMMU, which so far has neither been seen in the wild nor in academia. Support for additional target operating systems and other platforms like AMD and ARM is considered hereafter.

Eventually, HyperLeech should undergo a deeper evaluation against other approaches to make a measurable statement of its advantages regarding the analysis of environment-aware malware from a forensics perspective.

Acknowledgements

We would like to thank Ulf Frisk for his helpful comments and detailed insights into the PCILeech project. Furthermore, we thank Tobias Latzo for sharing his knowledge about co-processor-based injections.

Availability

As part of this project, we make our prototype implementation available upon request for research purposes.

References

[1] Ftdi drivers. http://www.ftdichip.com/Drivers/D3XX/FTD3XXLibrary_v1.2.0.6.zip, 2018.

[2] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 2–13. ACM, 2006.

[3] Ramtin Amin and Ulf Frisk. Pcileech. https://github.com/ufrisk/pcileech-fpga/tree/master/pciescreamer, 2019.

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS operating systems review*, 37(5):164–177, 2003.

[5] Darren Bilby. Low down and dirty: anti-forensic rootkits. *Proceedings of Ruxcon*, 2006.

[6] Jamie Butler. Dkom (direct kernel object manipulation). *Black Hat Windows Security*, 2004.

[7] Brian D Carrier and Joe Grand. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation*, 1(1):50–60, 2004.

[8] Jean-Christophe Delaunay. Practical dma attack on windows 10. https://www.synacktiv.com/posts/pentest/practical-dma-attack-on-windows-10.html, 2018.

[9] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. SPIDER: stealthy binary program instrumentation and debugging via hardware virtualization. In *Annual Computer Security Applications Conference, ACSAC '13, New Orleans, LA, USA, December 9-13, 2013*, pages 289–298, 2013.

[10] Artem Dinaburg, Paul Royal, Monirul I. Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, pages 51–62, 2008.

[11] Loïc Duflot, Yves-Alexis Perez, and Benjamin Morin. What if you can't trust your network card? In *Recent Advances in Intrusion Detection - 14th International Symposium, RAID 2011, Menlo Park, CA, USA, September 20-21, 2011. Proceedings*, pages 378–397, 2011.

[12] Mark Ermolov and Maxim Goryachy. How to hack a turned-off computer, or running unsigned code in intel management engine. *Black Hat Europe*, 2017.

[13] Aristide Fattori, Roberto Paleari, Lorenzo Martignoni, and Mattia Monga. Dynamic and transparent analysis of commodity production systems. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, pages 417–426, 2010.

[14] Linux Foundation. Openbmc - github. https://github.com/openbmc/openbmc, 2018.

[15] Ulf Frisk. Direct memory attack the kernel. *Proceedings of DEFCON*, 24, 2016.

[16] Ulf Frisk. Pcileech on macos. https://github.com/ufrisk/pcileech/wiki/Target-macOS, 2018.

[17] Ulf Frisk. Pcileech on linux. https://github.com/ufrisk/pcileech/wiki/PCILeech-on-Linux, 2019.

[18] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the Network and Distributed*

System Security Symposium, NDSS 2003, San Diego, California, USA, 2003.

[19] Ou George. Detecting the blue pill hypervisor rootkit is possible but not trivial. https://www.zdnet.com/article/detecting-the-blue-pill-hypervisor-rootkit-is-possible-but-not-trivial, 2006.

[20] Yasunori Goto. Kernel-based virtual machine technology. *Fujitsu Scientific and Technical Journal*, 47(3): 362–368, 2011.

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

[22] Takahiro Haruyama and Hiroshi Suzuki. One-byte modification for breaking memory forensic analysis. *Black Hat Europe*, 2012.

[23] Intel. Intel management engine. https://www.intel.com/content/www/us/en/support/articles/000008927/software/chipset-software.html, 2017.

[24] Intel. Intel 64 and ia-32 architectures software developer's manual volume 3 (3a, 3b, 3c & 3d): System programming guide, part 3. *Part*, 3, 2019.

[25] Bhushan Jain, Mirza Basim Baig, Dongli Zhang, Donald E. Porter, and Radu Sion. Sok: Introspections on trust and the semantic gap. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 605–620, 2014.

[26] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 128–138. ACM, 2007.

[27] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Vmm-based hidden process detection and identification using lycosid. In *Proceedings of the 4th International Conference on Virtual Execution Environments, VEE 2008, Seattle, WA, USA, March 5-7, 2008*, pages 91–100, 2008.

[28] Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch. Subvirt: Implementing malware with virtual machines. In *2006 IEEE Symposium on Security and Privacy (S&P 2006), 21-24 May 2006, Berkeley, California, USA*, pages 314–327, 2006. doi: 10.1109/SP.2006.38. URL https://doi.org/10.1109/SP.2006.38.

[29] Michael Kiperberg, Roee Leon, Amit Resh, Asaf Algawi, and Nezer Zaidenberg. Hypervisor-assisted atomic memory acquisition in modern systems. In *Proceedings of the 5th International Conference on Information Systems Security and Privacy, ICISSP 2019, Prague, Czech Republic, February 23-25, 2019*, pages 155–162, 2019.

[30] Igor Korkin and Satoshi Tanda. Detect kernel-mode rootkits via real time logging & controlling memory access. *CoRR*, abs/1705.06784, 2017.

[31] Nisha Lalwani, MB Chandak, and RV Dharaskar. Split personality malware: a security threat. In *IJCA Proc. National Conf. Innovative Paradigms in Engineering and Technology (NCIPET 2012)*, number 14, 2012.

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

[33] Tobias Latzo, Julian Brost, and Felix Freiling. Bmcleech: Introducing stealthy memory forensics to bmc. *Digital Investigation*, 2020.

[34] Tamas K. Lengyel, Steve Maresca, Bryan D. Payne, George D. Webster, Sebastian Vogl, and Aggelos Kiayias. Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system. In *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014, New Orleans, LA, USA, December 8-12, 2014*, pages 386–395, 2014.

[35] Lionel Litty, H. Andrés Lagar-Cavilla, and David Lie. Hypervisor support for identifying covertly executing binaries. In Paul C. van Oorschot, editor, *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA*, pages 243–258. USENIX Association, 2008.

[36] Carsten Maartmann-Moe. Inception. http://www.breaknenter.org/projects/inception/, 2011.

[37] A. Theodore Markettos, Colin Rothwell, Brett F. Gutstein, Allison Pearce, Peter G. Neumann, Simon W. Moore, and Robert N. M. Watson. Thunderclap: Exploring vulnerabilities in operating system IOMMU protection via DMA from untrustworthy peripherals. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*, 2019.

[38] Alex Markuze, Adam Morrison, and Dan Tsafrir. True IOMMU protection from DMA attacks: When copy is faster than zero copy. In *Proceedings of the Twenty-First International Conference on Architectural Support*

for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016, pages 249–262, 2016.

[39] Lorenzo Martignoni, Aristide Fattori, Roberto Paleari, and Lorenzo Cavallaro. Live and trustworthy forensic analysis of commodity production systems. In *Recent Advances in Intrusion Detection, 13th International Symposium, RAID 2010, Ottawa, Ontario, Canada, September 15-17, 2010. Proceedings*, pages 297–316, 2010.

[40] Benoît Morgan, Eric Alata, Vincent Nicomette, and Mohamed Kaâniche. IOMMU protection against I/O attacks: a vulnerability and a proof of concept. *J. Braz. Comp. Soc.*, 24(1):2:1–2:11, 2018.

[41] Anh M. Nguyen, Nabil Schear, HeeDong Jung, Apeksha Godiyal, Samuel T. King, and Hai D. Nguyen. MAVMM: lightweight and purpose built VMM for malware analysis. In *Twenty-Fifth Annual Computer Security Applications Conference, ACSAC 2009, Honolulu, Hawaii, USA, 7-11 December 2009*, pages 441–450, 2009.

[42] Ralph Palutke and Felix C. Freiling. Styx: Countering robust memory acquisition. *Digital Investigation*, 24: S18–S28, 2018.

[43] Fabien Périgaud, Alexandre Gazet, and Joffrey Czarny. Subverting your server through its bmc: the hpe ilo4 case. *Recon Brussels*, 2018.

[44] Nick L Petroni Jr, Timothy Fraser, Jesus Molina, and William A Arbaugh. Copilot-a coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium*, pages 179–194. San Diego, USA, 2004.

[45] Jonas Pfoh, Christian A. Schneider, and Claudia Eckert. Nitro: Hardware-based system call tracing for virtual machines. In *Advances in Information and Computer Security - 6th International Workshop, IWSEC 2011, Tokyo, Japan, November 8-10, 2011. Proceedings*, pages 96–112, 2011.

[46] Sergej Proskurin, Julian Kirsch, and Apostolis Zarras. Follow the whiterabbit: Towards consolidation of on-the-fly virtualization and virtual machine introspection. In *ICT Systems Security and Privacy Protection - 33rd IFIP TC 11 International Conference, SEC 2018, Held at the 24th IFIP World Computer Congress, WCC 2018, Poznan, Poland, September 18-20, 2018, Proceedings*, pages 263–277, 2018.

[47] Sergej Proskurin, Tamas K. Lengyel, Marius Momeu, Claudia Eckert, and Apostolis Zarras. Hiding in the shadows: Empowering ARM for stealthy virtual machine introspection. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*, pages 407–417, 2018.

[48] T Ptacek, Nate Lawson, and P Ferrie. Don't tell joanna, the virtualized rootkit is dead. *Black Hat*, 2007.

[49] Zhengwei Qi, Chengcheng Xiang, Ruhui Ma, Jian Li, Haibing Guan, and David S. L. Wei. Forenvisor: A tool for acquiring and preserving reliable data in cloud live forensics. *IEEE Trans. Cloud Computing*, 5(3):443–456, 2017.

[50] Junghwan Rhee, Ryan Riley, Dongyan Xu, and Xuxian Jiang. Defeating dynamic data kernel rootkit attacks via vmm-based guest-transparent monitoring. In *Proceedings of the The Forth International Conference on Availability, Reliability and Security, ARES 2009, March 16-19, 2009, Fukuoka, Japan*, pages 74–81. IEEE Computer Society, 2009.

[51] Paul Royal. Alternative medicine: The malware analyst's blue pill. *Black Hat USA*, 2008.

[52] Joanna Rutkowska. Subverting vistatm kernel for fun and profit. *Black Hat Briefings*, 2006.

[53] Joanna Rutkowska. Beyond the cpu: Defeating hardware based ram acquisition. *Proceedings of BlackHat DC*, 2007, 2007.

[54] Joanna Rutkowska. Security challenges in virtualized environments. In *Proceedings RSA conference 2008*, 2008.

[55] Joanna Rutkowska and Alexander Tereshkin. Isgameover () anyone. *Black Hat, USA*, 2007.

[56] Monirul I. Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. Secure in-vm monitoring using hardware virtualization. In *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, pages 477–487, 2009.

[57] Jiangyong Shi, Yuexiang Yang, Chengye Li, and Xiaolei Wang. SPEMS: A stealthy and practical execution monitoring system based on VMI. In *Cloud Computing and Security - First International Conference, ICCCS 2015, Nanjing, China, August 13-15, 2015. Revised Selected Papers*, pages 380–389, 2015.

[58] Sherri Sparks and Jamie Butler. Shadow walker: Raising the bar for rootkit detection. *Black Hat Japan*, 11(63): 504–533, 2005.

[59] Johannes Stüttgen and Michael Cohen. Anti-forensic resilient memory acquisition. *Digital investigation*, 10: S105–S115, 2013.

[60] Johannes Stüttgen and Michael Cohen. Robust linux memory acquisition with minimal target impact. *Digital Investigation*, 11(1):S112–S119, 2014.

[61] Jacob Torrey. More shadow walker: Tlb-splitting on modern x86. *Blackhat USA*, 2014.

[62] Stefan Vömel and Felix C Freiling. A survey of main memory acquisition and analysis techniques for the windows operating system. *Digital Investigation*, 8(1):3–22, 2011.

[63] Carsten Willems, Ralf Hund, and Thorsten Holz. Cxpinspector: Hypervisor-based, hardware-assisted system monitoring. *Ruhr-Universitat Bochum, Tech. Rep*, page 12, 2013.

[64] Lok-Kwong Yan, Manjukumar Jayachandra, Mu Zhang, and Heng Yin. V2e: combining hardware virtualization and software emulation for transparent and extensible malware analysis. *ACM Sigplan Notices*, 47(7):227–238, 2012.

[65] Miao Yu, Zhengwei Qi, Qian Lin, Xianming Zhong, Bingyu Li, and Haibing Guan. Vis: Virtualization enhanced live forensics acquisition for native system. *Digital Investigation*, 9(1):22–33, 2012.

[66] Patrycjusz Zdzichowski, Michal Sadlon, Teemu Uolevi Väisänen, Alvaro Botas Munoz, and Karina Filipczak. Anti-forensic study. *NATO CCDCOE (NATO Cooperative Cyber Defence Centre of Excellence)*, 2015.

[67] Lei Zhang, Lianhai Wang, Ruichao Zhang, Shuhui Zhang, and Yang Zhou. Live memory acquisition through firewire. In *Forensics in Telecommunications, Information, and Multimedia - Third International ICST Conference, e-Forensics 2010, Shanghai, China, November 11-12, 2010, Revised Selected Papers*, pages 159–167, 2010.

[68] Ning Zhang, Kun Sun, Wenjing Lou, Yiwei Thomas Hou, and Sushil Jajodia. Now you see me: Hide and seek in physical address space. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15, Singapore, April 14-17, 2015*, pages 321–331, 2015.

[69] Dino A Dai Zovi. Hardware virtualization rootkits. *Black Hat 2006, August*, 2006.