The SEVerESt Of Them All: Inference Attacks Against Secure Virtual Enclaves

Jan Werner
UNC Chapel Hill
jjwerner@cs.unc.edu

Joshua Mason
U. Illinois
joshm@illinois.edu

Manos Antonakakis
Georgia Tech
manos@gatech.edu

Michalis Polychronakis
Stony Brook University
mikepo@cs.stonybrook.edu

Fabian Monrose
UNC Chapel Hill
fabian@cs.unc.edu

ABSTRACT

The success of cloud computing has shown that the cost and convenience benefits of outsourcing infrastructure, platform, and software resources outweigh concerns about confidentiality. Still, many businesses resist moving private data to cloud providers due to intellectual property and privacy reasons. A recent wave of hardware virtualization technologies aims to alleviate these concerns by offering encrypted virtualization features that support data confidentiality of guest virtual machines (*e.g.,* by transparently encrypting memory) even when running on top untrusted hypervisors.

We introduce two new attacks that can breach the confidentiality of protected enclaves. First, we show how a cloud adversary can judiciously inspect the general purpose registers to unmask the computation that passes through them. Specifically, we demonstrate a set of attacks that can precisely infer the executed instructions and eventually capture sensitive data given only indirect access to the CPU state as observed via the general purpose registers. Second, we show that even under a more restrictive environment — where access to the general purpose registers is no longer available — we can apply a different inference attack to recover the structure of an unknown, running, application as a stepping stone towards application fingerprinting. We demonstrate the practicality of these inference attacks by showing how an adversary can identify different applications and even distinguish between versions of the same application and the compiler used, recover data transferred over TLS connections within the encrypted guest, retrieve the contents of sensitive data as it is being read from disk by the guest, and inject arbitrary data within the guest. Taken as a whole, these attacks serve as a cautionary tale of what can go wrong when the state of registers (*e.g.,* in AMD's SEV) and application performance data (*e.g.,* in AMD's SEV-ES) are left unprotected. The latter is the first known attack that was designed to specifically target SEV-ES.

ACM Reference Format:
Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. 2019. The SEVerESt Of Them All: Inference Attacks Against Secure Virtual Enclaves. In *ACM Asia Conference on Computer and Communications Security (AsiaCCS '19), July 9–12, 2019, Auckland, New Zealand.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3321705.3329820

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
AsiaCCS '19, July 9–12, 2019, Auckland, New Zealand
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6752-3/19/07…$15.00
https://doi.org/10.1145/3321705.3329820

1 INTRODUCTION

Of late, the need for a Trusted Execution Environment has risen to the forefront as an important consideration for many parties in the cloud computing ecosystem. Cloud computing refers to the use of on-demand networked infrastructure software and capacity to provide resources to customers [52]. In today's marketplace, content providers desire the ability to deliver copyrighted or sensitive material to clients without the risk of data leaks. At the same time, computer manufacturers must be able to verify that only trusted software executes on their hardware, and OS vendors need guarantees that no malicious code executes upon boot. Likewise, clients that use cloud services for their touted costs and security benefits expect confidentiality of the information stored on cloud servers.

Fortunately, some of these requirements can be met with existing offerings. For example, the ARM Trust Zone™ technology allows customers to build robust Digital Rights Management systems, Secure Boot technologies (*e.g.,* Intel Trusted Execution Technology [17] and the Trusted Platform Module [49]) guarantee that only trusted software is loaded, and so-called Trusted Path mechanisms [38] provide a secure interface for inputting sensitive data. However, until recently, no practical solutions were available for ensuring the confidentiality of cloud computation from the cloud provider itself. Indeed, inquisitive cloud providers can readily inspect and modify customer's information using virtual machine introspection [22], and so to alleviate that threat, customers typically resort to business agreements to protect their assets.

Within the cloud computing arena, virtualization is the de facto technology used to provide isolation of tenants. More specifically, hypervisors are used to provide both temporal and spatial separation of virtual machines (VMs) for different customers running on a single cloud instance. While the advent of hypervisor technology has been a boon for cloud computing, its proliferation comes with several risks. For one, bugs in the hypervisor can undermine the isolation and integrity properties offered by these technologies and thus clients utilizing the cloud infrastructure must place full trust in the cloud provider.

To confront the problem of having fully trusted hypervisors, in late 2016, Advanced Micro Devices (AMD) announced new security extensions [27]. In particular, their Secure Memory Encryption subsystem allows for full system memory encryption and aims to ensure data confidentiality against physical attacks such as cold boot and DMA attacks [21]. A second extension, dubbed Secure Encrypted Virtualization (SEV), couples memory encryption with virtualization extensions that allows for per virtual machine memory encryption. SEV takes a bold step forward by aiming to protect virtual machines not only from physical attacks, but from other virtual machines and *untrusted hypervisors.* A third extension, coined Secure Encrypted Virtualization with Encrypted State (SEV-ES) [26], builds upon SEV and purportedly allows for encryption of all processor registers when the VM is paused. These new extensions provide a private, per-virtual machine memory encryption solution that is performed entirely in hardware, independently from the virtual machine manager. Accepting the importance of cloud confidentiality, some cloud providers have recently announced the availability of such security protections on their platforms[1]. Somewhat at odds with this acceptance, however, is the fact that while cloud providers stipulate that the customer is responsible for complying with an acceptable use policy (AUP), they reserve the right to review the applications and data for compliance with the AUP.

These new cloud platforms are built around a threat model where "an attacker is assumed to have access to not only execute user level privileged code on the target machine, but can potentially execute malware at the higher privileged hypervisor level as well" [27]. The conjecture is that even under these circumstances, secure encrypted virtualization provides assurances to help protect the guest virtual machine code and data from an attacker. In this paper, we investigate the extent to which existing encrypted virtualization technologies do in fact provide the desired security guarantees in term of a guest VM's data confidentiality. We argue that such an investigation is particularly timely, as the complexities of hardware implementations coupled with software deployment challenges and push-to-market pressures, have led to commercially available technologies (like SEV) leaving general purpose registers unprotected. Although such a design decision immediately raised concerns about the potential for data leakage [32] when SEV was first announced, these concerns were largely dismissed due to the seemingly challenging nature of mounting such an attack — namely that a malicious hypervisor would still be completely "blind" in terms of the guest VM's memory state, thereby making such attacks difficult to pull off in practice.

Unfortunately, we show that this is not the case. As we demonstrate later, this design decision opens the door to a new class of attacks that allow a cloud adversary to fully breach the confidentiality of a protected guest VM by judiciously inspecting the state of the underlying general purpose registers. To that end, we introduce a new class of CPU *register inference attacks* that can precisely infer a victim VM's stream of executed instructions, and eventually leak sensitive data given only indirect access to the CPU state as observed via the general purpose registers. Using SEV as our main use case, we demonstrate the practicality of this new class of inference attacks by showing how an adversary can efficiently recover data being communicated over TLS connections within the encrypted guest, retrieve the contents of sensitive data as it is being read from disk by the guest, and inject arbitrary data within the guest via Iago-style attacks [8], *without any prior knowledge about the memory state of the guest VM.* We believe such attacks directly apply to any security designs that leave register contents unprotected.

Additionally, we present a novel application fingerprinting technique that allows a cloud adversary, or malicious tenant, to precisely identify the applications running in the SEV-ES protected machine, including details such as the version and the compiler used to build the target application. To do so, we introduce a new binary-level signature that captures the uniqueness of the layout of functions in an application, demonstrate how to efficiently collect the data from a performance measurement subsystem, and use the collected data to perform matching on a data store of target applications (*e.g.,* AUP forbidden services like Bitcoin mining).

In summary, our work makes the following contributions:

1. We introduce a new class of register inference attacks for unveiling information in secure enclaves wherein the adversary only has intermittent access to the CPU registers.
2. We present concrete implementations and empirical analyses of attacks (*e.g.,* on SEV) that demonstrate how a cloud adversary (or a tenant that exploits bugs in the hypervisor) can unveil sensitive data of protected VMs.
3. We introduce a new fingerprinting technique for precisely identifying applications running in secure enclaves that do *not* leak register state. For that, we leverage application performance data to uncover structural properties of applications running in the guest VMs.
4. We suggest mitigations for the uncovered weaknesses, as well as directions for future work.

Our attacks not only validate the security community's speculation that leaving general purpose registers unencrypted may eventually lead to data leakage, but highlight the powerful nature of the attacks that become possible in the context of secure virtualization technologies when only limited information is available to an adversary. Additionally, we show that designing secure virtualization platforms is far more difficult than it seems — especially in terms of cloud confidentiality. Our ultimate goal is to raise awareness of this new class of inference attacks, with the hope that by doing so, our work will better inform future design decisions.

2 BACKGROUND

For pedagogical reasons, we briefly recap the architecture available by AMD as it is representative of the state of the art in this domain. Specifically, "Secure Encrypted Virtualization (SEV) integrates main memory encryption capabilities with the existing AMD-V virtualization architecture to support encrypted virtual machines. Encrypting virtual machines can help protect them not only from physical threats but also from other virtual machines or even the hypervisor itself" [27]. The Secure Encrypted Virtualization Encrypted State (SEV-ES) in addition to encrypting the main memory, protects the guest register state from the hypervisor. When SEV-ES is enabled, guest virtual machines are granted control over the exits to the hypervisor and the data that is provided during the exit.

The Key management is handled by the Platform Security Processor (PSP), thus software running on the main processor (*i.e.,* hypervisor) can not access the memory encryption key. The PSP is responsible for providing guest measurements during VM provisioning, secure migration of the VM and guest VM debugging.

[1]https://arstechnica.com/gadgets/2017/09/azure-confidential-computing-will-keep-data-secret-even-from-microsoft/

Table 1: Comparison of secure enclave technologies and attacks.

Technology	SGX	SEV	SEV-ES
Protected zone	Enclave in userspace	Virtual Machine	Virtual Machine
Scope	No access to registers	Full access to guest registers	Guest controls access to registers
Control of state transition	Host controls the enclave	Hypervisor controls VMEXITs	Guest controls most of VMEXITs
Available information	Software executing in the enclave, page faults	Software executing in the enclave [36], [24], Register state, page faults, debug events	Application performance data (virtual addr, physical addr, instruction type (branch, load/store))
Attacks	Data in the enclave [41], [30], [31], [53], [7]	Recovery of specific data[36], [24] Replay of code [24] Recovery of instructions executed [**this paper**]	Structure of an application [**this paper**]

For efficiency, memory encryption is performed using a high performance AES engine in dedicated hardware in the memory controllers. Each virtual machine is tagged with an associated encryption key, and data is restricted to only the VM using that tag. It is expected that the guest owner provides the guest image to the cloud system. The firmware assists in launching the guest and provides a measurement back to the guest owner. If the guest owner deems this measurement correct, they in turn provide additional secrets (such as a disk decryption key) to the running guest to allow it to proceed with start-up. The key is provided via a secure key management interface to ensure that the hypervisor never has access to it. Confidentiality of the guest is accomplished by encrypting memory with a key that only the firmware knows. The management interface does not allow the memory encryption key or any other secret state to be exported outside of the firmware without properly authenticating the recipient. This prevents the hypervisor from gaining access to the keys[2].

2.1 Register Inference Attacks

To re-cap, when SEV is enabled, the security processor is used to automatically encrypt and decrypt the contents of memory on transitions between the guest and the hypervisor. All that is left unencrypted are general purpose registers in the virtual machine control block and DMA pages used by the virtual input/output devices. To see why this is a cause of concern, let us first assume the expected deployment model where: *the owner of the virtual machine verifies the platform measurements, the disk image of the VM is encrypted and cannot be read by the hypervisor, and the guest policy is set to prevent the debug access to VM memory.* Additionally, the hypervisor has access to shared memory regions (*e.g.,* input/output data buffers), it can force the guest to exit, and it has access to the Virtual Machine Control Block (VMCB) that includes unencrypted processor registers (general purpose and machine specific).

With that in mind, recall that the CPU fetches instructions from memory locations indicated by the instruction pointer, then decodes and executes them. Optionally, the results can be stored in memory. Different types of instructions (*e.g.,* arithmetic, logical, and storage) all operate on general purpose registers and memory locations. Special classes of instructions such as floating point arithmetic or hardware assisted encryption, operate on dedicated registers.

In Section 4, we show the security afforded by designs that leave register contents unprotected can be undermined by judiciously inspecting the general purpose registers and unmasking the computation that passes through them. By passively observing changes in the registers, an adversary can recover critical information about activities in the encrypted guest. Naïvely, one could do this by single stepping a target process from within the hypervisor, but doing so would incur a *significant* performance penalty that would be easily noticeable in the guest. Moreover, the amount of data collected (*i.e.,* register contents) would quickly become overwhelming.

2.2 Structural Inference Attacks

When SEV-ES is enabled, the register state in the Virtual Machine Control Block *is no longer available.* SEV-ES not only encrypts but also protects the integrity of the VMCB, thus preventing attacks on the register state. A new structure called Guest Hypervisor Control Block (GHCB) [4] acts as an intermediary between the guest and the hypervisor during hypercalls. The guest dictates, via a policy, what information is shared in the GHCB. Furthermore, from a design perspective, VMEXITs are classified as either Automatic (AE) or Non-Automatic (NAE); AE events do not require exposing any state from the guest and trigger an immediate control transfer to the hypervisor. Performance measurement events, such as the delivery of an interrupt when the data is ready, are automatic events. In Section 4.3, we show how one can use data provided by the Instruction Based Sampling (IBS) subsystem (*e.g.,* to learn whether an executed instruction was a branch, load, or store) to identify the applications running within the VM. Intuitively, one can collect performance data from the virtual machine and match the observed behavior to known signatures of running applications.

Before delving into the particulars of our attacks, we first review some pertinent related work to help set the broader context. For ease of presentation, Table 1 provides a high-level overview of the protection provided by commercial offerings, the information leaked under each offering, and the types of attacks known to date. As noted we assume the ability to control the hypervisor running the secured VMs, but not any prior knowledge of the software running inside the VMs. The attack against SEV protected VMs requires access to general purpose registers (in VMCB), as well as the ability to control second level translation page faults. The attack against SEV-ES only assumes the availability of information from the performance measuring subsystem (*i.e.,* IBS).

3 RELATED WORK

Side-Channels. To date, numerous attacks (*e.g.,* [29, 30, 33, 35, 50]) have been proposed for leaking information across security domains by leveraging architectural or micro-architectural artifacts.

[2]This assumes a correct implementation of the PSP and so the vulnerabilities presented by CTS-LABS [12] are out of scope.

Lee et al. [31], for instance, demonstrated the feasibility of fine grained control flow attacks in an enclave. The attack takes advantage of the fact that SGX does not clear the branch history when switching security boundaries. Like other attacks [7, 41, 45, 58] on SGX, the adversary model assumes the attacker knows the possible flows of a target enclave program (*e.g.,* by statically or dynamically analyzing its source code or binary).

Using similar knowledge, Xu et al. [58] introduce so-called controlled-channel attacks to extract information from victim enclaves. Such attacks exploit secret-dependent control flow and data accesses in legacy applications. The attacks are based on the observation that a processor in enclave mode accesses unprotected page table memory during the address translation process, and so a page table walk can be used to identify which pages were accessed. Specifically, they rely on sequences of page faults to identify specific memory addresses, and show that the page fault side channel is strong enough to extract sensitive data from enclave applications.

Wang et al. [53] provide a review of memory and cache side channels, and propose a series of cache-line attacks related to address translation in CPU hardware. Their attacks achieve spatial granularity via cross-enclave PRIME+PROBE [20] attacks or cross-enclave shared DRAM attacks (*e.g.,* [28, 40, 57]) to extract sensitive information. More recently, Kocher et al. [29] and Lipp et al. [33] introduce ingenious micro-architectural attacks that trick the processor into speculatively executing instructions that ultimately leak information about a victim's memory address space. The interested reader is referred to [6, 34, 46] for excellent surveys. As articulated in Table 1, these attacks are unrelated to what we study herein.

Attacks on SEV. Most germane are the ideas presented by Hetzelt and Buhren [24] and Sharkey [44]. Hetzelt and Buhren [24] analyzed the design documents for SEV and posit a series of attacks. The most powerful of their attacks tries to force leakage of information by introducing an interruption of the guest execution after protected data has been transferred from an attacker controlled memory location into an unencrypted register. The authors show how the gadgets used to force the `vmexit` can be located via static analysis, and used later in an online attack. Essentially, Hetzelt and Buhren [24] argue that an adversary can perform a *linear sweep of memory* where the contents of the VM are dumped by forcing a `vmexit` to read the data moved to the unencrypted register.

Although the attack is plausible, it has several practical limitations. For one, it assumes the adversary has in-depth knowledge of the guest's operating system (*e.g.,* the kernel binary layout). In fact, lack of knowledge of exact offsets renders the attacks [24, Section 5.1] infeasible. Most critically, the attack requires intricate knowledge of the memory allocation process in order to allocate an unencrypted page in the guest or to modify an encrypted page to remove the C-bit [27] that enables memory encryption.

Sharkey [44] discussed a similar idea at BlackHat, but the techniques for undermining SEV either bypass the load-time integrity checks (*e.g.,* to install a rootkit) or intercept the AES-NI instructions (to read the register contents) by generating an exception anytime an AES-NI instruction is executed. The adversarial model also assumes no encrypted storage or attestation is in place, which *does not* conform to AMD's envisioned deployment model.

Alternatively, Du et al. [14] explored the possibility of chosen ciphertext attacks against SEV, based on theoretical weaknesses in the tweak-based algorithm suggested by AMD as a replacement to AES-CTR mode. They argue that attacks can be designed conceptually based on the assumption that *i)* data integrity protection is not provided in SEV and *ii)* the tweak algorithm uses host physical addresses instead of VM physical addresses;[3] the combination of which allows an adversary to swap the VM addresses of two encrypted pages and perform chosen ciphertext attacks under specific conditions. As a mitigation, the authors recommend the use of a different key derivation function (namely NIST SP 800-108). Like the work of Hetzelt and Buhren [24], their attack was only a simulation and not on the SEV-enabled hardware that became available in late 2017. Most recently, Morbitzer et al. [36, 37] explored the idea of using the applications running within the protected enclave to leak the data from the VM. This attack hinges on the presence of an application that serves data from the VM to the outside (*e.g.,* a web server) and uses the second level address translation to replace the memory pages of the content to be served, where the chosen pages are those an adversary wants to leak.

Unlike these works, we present a comprehensive technique for identifying the instructions executed in the encrypted guest given only a trace of changes of general purpose registers. Our analyses are on the real hardware and we pay particular attention to generalizable techniques (*e.g.,* that go beyond simply peeking into AES-NI instructions [44]) that can be stealthily performed.

Attacks on SEV-ES. To the best of our knowledge, there are no attacks specifically designed for SEV-ES. Although the attack on SEV by Morbitzer et al. [36, 37] does not depend on any state that would be encrypted under SEV-ES, success of that attack hinges on the presence of a remote communication service running in the target VM. Specifically, like Du et al. [14], Morbitzer et al. [37] take advantage of the fact that since (*i*) SEV-encrypted VMs lack integrity protection, and (*ii*) the hypervisor is responsible for second-level address translation, a malicious or compromised hypervisor could leverage the communication service to learn host physical address mappings in main memory, change the memory layout, and subsequently leak the contents of memory. Of course, re-mapping the memory for applications in flight can easily lead to crashing the affected applicatons. We have no such limitation. Moreover, such an attack is far from stealthy, requiring hundreds of thousands of requests over several hours, during which any visitor requesting data from the attacked service would notice the nonsensical responses [37]. Our attacks, by contrast, can be perpetrated with only a handful of requests, and incur little to no user-perceived delay.

4 APPROACH

One challenge with our register inference attacks is that millions of data points might need to be collected and processed. To alleviate that, we take a white-box approach. The intuition is that since only a small part of a target process' computation needs to be closely monitored, we can first identify that part using coarse-grained monitoring, and once the computation of interest is about to begin, switch to fine-grained monitoring. The key is knowing when to switch modes.

As we show later, white-box analysis can be used to build profiles of applications of interest (*e.g.,* web servers) in the guest by

[3]We have been unable to confirm this is indeed true in SEV.

monitoring system calls within the hypervisor. Having done so, white-box analysis can further be used to hone in on just the right parts in the execution of the target application and perform limited introspection of the registers. For example, after identifying an application as a web server, one can focus on routines responsible for sending and receiving network traffic. For that, the steps required to recover information in the encrypted guest include:

(1) Identifying the guest VM operating system. This can be achieved in several ways, for example, by inspecting the usage of Model Specific Registers (different operating systems tend to use different ranges of virtual memory addresses for kernel entry points) or identifying system call sequences (*e.g.*,`execve()`,`brk()` are the first calls invoked by a new Linux process).
(2) Applying OS specific heuristics to identify targets (for example, using the sequence: `[socket(),bind(),listen()]` to identify network servers).
(3) Profiling and matching applications of interest at runtime, *e.g.*, using coarse-grained system call tracking.
(4) Using a combination of instruction recovery techniques and white-box analysis to determine the best point to trigger the inspection of critical code and recover the plaintext (*e.g.*, the messages received from a victim server).

Any mechanism to selectively inspect a target process (*i.e.*, Step 4) must be done with stealth. We refer to this process as hyper-stepping (§4.2). As a part of our exploration of the feasibility of conducting inference attacks on SEV-enabled platforms, we investigated several approaches to trapping system calls made in the target VM, including using the Monitor Trap Flag to exit to hypervisor, using second level translation (*e.g.*, Intel's EPT and AMD's RVI hardware assisted paging (HAP) page faults), disabling the system call instruction, and using hardware debug registers.

Using the Monitor Trap Flag is not an option as it is unavailable in the AMD architecture. Although HAP is an option, we discarded it because we wanted to limit the amount of exits to the hypervisor associated with handling of a system call. Additionally, the use of the invalid opcode exception implies the ability to read the opcode of the failing instruction, but that is unavailable when SEV is enabled. Thus, we settled on using hardware debug registers.

Since OS and application fingerprinting using system calls has been the subject of much prior work (*e.g.*, [16, 42, 54]), we focus the remaining discussion on steps 3 and 4.

4.1 Efficiently tracking system calls

To track guest OS system calls from within the hypervisor, one needs to inspect the guest state at the entry and exit points of the system call. Inspection is needed not only to determine which system call is being issued and its parameters, but also the result of the system call. Modern operating systems use fast system call instructions (`SYSENTER` for 32 bit and `SYSCALL` for 64 bit OSes) to transition to the kernel and execute the system calls. The virtual address of the kernel entry point is specified in a special Model Specific Register (MSR), and the virtual address of the system call return address is stored in the `RCX` register during the `SYSCALL` instruction invocation. Thus, using two hardware debug registers one can effectively trace guest OS system calls. Note that in contemporary systems, the hypervisor ultimately controls the access to the

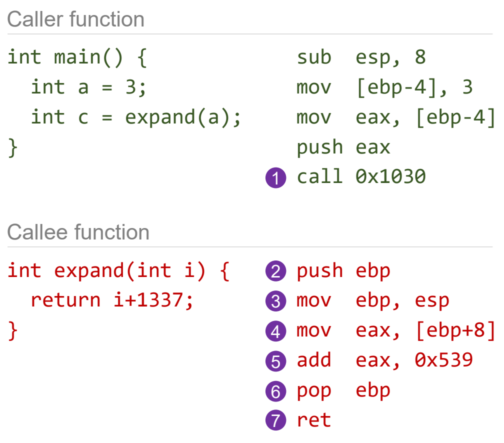

Caller function
```
int main() {              sub  esp, 8
  int a = 3;              mov  [ebp-4], 3
  int c = expand(a);      mov  eax, [ebp-4]
}                         push eax
                        ① call 0x1030
```
Callee function
```
int expand(int i) {     ② push ebp
  return i+1337;        ③ mov  ebp, esp
}                       ④ mov  eax, [ebp+8]
                        ⑤ add  eax, 0x539
                        ⑥ pop  ebp
                        ⑦ ret
```

Figure 1: C source with its corresponding assembly code snippet.

debug registers, and so if the guest OS attempts to inspect or use them, the hypervisor can still retain control and evade detection.

4.2 Hyper-stepping

Once the target has been identified using system call tracking, the next task is to observe and control the execution of the critical code. In our case, the code of the critical section is executed an instruction at a time — essentially, the hypervisor acts as a debugger single stepping the execution in the guest VM. Since the memory of the guest is encrypted, the hypervisor single stepping execution in the guest is not capable of reading the process memory and disassembling the executed instruction. The only available information is the outcome of the execution as observed in the general purpose registers. Yet, that information can significantly aid our understanding of the executed instruction. Equally important is the fact that the hypervisor is capable of modifying the guest's state by carefully manipulating the general purpose registers. In particular, modifying the contents of registers (§5) enables sophisticated attacks against the guest VM.

The success of our attacks hinges on our ability to correctly identify which instructions were executed in the guest. In certain cases, we require additional auxiliary information, in particular, knowledge of what specific registers (*e.g.*, `EDI` vs `ECX`) were accessed and the type (read or write) of memory access. Our approach for gathering that insight is presented next.

4.2.1 Unmasking instructions "in the dark". The high-level idea centers on the observation that by inspecting the state of the CPU registers one can try to infer the executed instructions by mapping the outcomes seen in the registers to known instruction semantics. For instance, consider the simple C program and the trace of its execution shown in Figure 1.

For presentation purposes, we chose a simplistic program that adds a constant value to an integer and returns the result. The depicted execution trace omits the setup and the tear down of the program. This trace represents the instructions executed by the CPU, not the instructions in the program image (*i.e.*, in memory or on disk). For the discussion that follows, we start our analysis with the `call` instruction in the caller (reference line ①) in Figure 1. The depiction in Figure 2 illustrates how one can reconstruct the execution given the luxury of observing seven steps. We denote an observation window as a pair of two directly connected register sets. Each numbered step represents an execution of an instruction, and the register sets represent the CPU state.

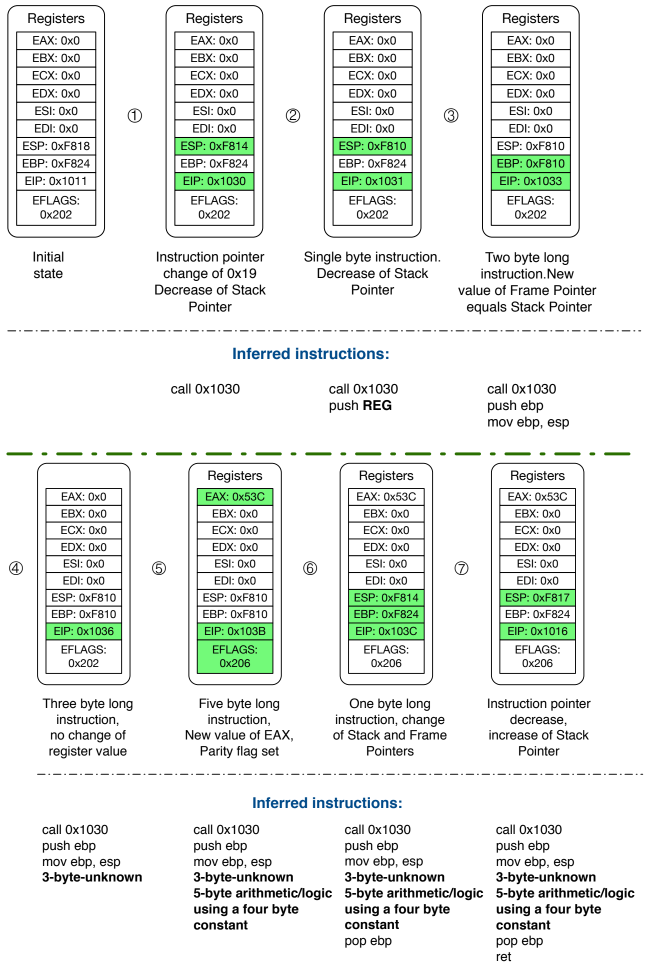

Registers
EAX: 0x0
EBX: 0x0
ECX: 0x0
EDX: 0x0
ESI: 0x0
EDI: 0x0
ESP: 0xF818
EBP: 0xF824
EIP: 0x1011
EFLAGS: 0x202
Initial state
①
ESP: 0xF814
EIP: 0x1030
Instruction pointer change of 0x19 Decrease of Stack Pointer
②
ESP: 0xF810
EIP: 0x1031
Single byte instruction. Decrease of Stack Pointer
③
EBP: 0xF810
EIP: 0x1033
Two byte long instruction.New value of Frame Pointer equals Stack Pointer
Inferred instructions:
call 0x1030
call 0x1030
push REG
call 0x1030
push ebp
mov ebp, esp
④
EIP: 0x1036
Three byte long instruction, no change of register value
⑤
EAX: 0x53C
EIP: 0x103B
EFLAGS: 0x206
Five byte long instruction, New value of EAX, Parity flag set
⑥
ESP: 0xF814
EBP: 0xF824
EIP: 0x103C
One byte long instruction, change of Stack and Frame Pointers
⑦
ESP: 0xF817
EIP: 0x1016
Instruction pointer decrease, increase of Stack Pointer
Inferred instructions:
call 0x1030
push ebp
mov ebp, esp
3-byte-unknown
5-byte arithmetic/logic using a four byte constant
pop ebp
ret

Figure 2: Example showing the inferred instructions at each stage.

In step ① in Figure 2, the advancement of the instruction pointer (register EIP) by more than 15 bytes indicates the presence of a control flow instruction. Given the decrease of the Stack Pointer (register ESP) we can identify this instruction as a `call` instruction. In step ②, the instruction pointer increases by one and the value of Stack Pointer is decreased by 4. This allows us to identify the instruction as a `push`. However, we cannot determine which register has been pushed onto the stack just yet. In step ③, the EIP increment indicates an execution of an instruction that is two bytes long. Inspection of the values of registers reveals that after the instruction is executed, the stack frame and stack pointer have the same value. This implies an assignment of the value from the stack pointer register to stack frame (register EBP) — *i.e.*, a `mov` instruction. Hence, we can assume that the register pushed onto the stack in step ② was the Frame Pointer.

Notice that in step ④, the only observed change is the advancement of the instruction pointer by three. This behavior is indicative of a load/store/arithmetic/logical type of instruction where both operands refer to registers. The observation of the unchanged Flag register suggests load/store type of instruction. In step ⑤, the instruction pointer advances by five bytes, the accumulator register EAX has a new value 0x53c, and the Parity Bit of the Flag Register is set. Given the computed length and the instruction outcome, we can surmise that this is due to loading a constant value. Change of the Parity Bit indicates an arithmetic operation. In step ⑥, a single byte instruction that decreases the value of the Stack Pointer and the change of the value of the Frame Pointer uniquely identify the instruction as a `pop` of the Frame Pointer register. This confirms the identification of the instruction in step ②. In step ⑦ the decrease of the instruction pointer indicates a control flow instruction. The increase of the stack pointer means that this is a `ret` instruction.

In that example, we can uniquely identify five out of seven instructions. We also narrowed down the set of possible instructions executed in steps ④ and ⑤. Assuming that the program follows standard C calling conventions, identification of the instruction in step ④ is easy; moving the function argument from the stack to register EAX in this case does not change the value of the register, because in the caller the function argument was first placed in the register EAX, then pushed onto the stack. Unfortunately, without additional information, an exact identification of the instruction executed in step ⑤ is not possible because several instructions (*e.g.*, `add, sub, or, xor`) are all likely based on the observed trace.

4.2.2 *Memory access identification.*

In the case of instructions that produce no observable change of the registers, one needs additional insights regarding the characteristics of the instruction. For that, the hypervisor can attempt to distinguish the types of memory access triggered by the guest. That can be done by intercepting the page fault exception and forcing the guest to re-execute the instruction with specific general purpose register values aimed to trigger a memory access violation. Recall that page fault handlers provide both the virtual address where the fault occurred as well as the error code indicating the type of access. Thus, forcing the guest to re-execute the instruction provides an opportunity to determine the computation of the *effective address* used by the instruction, that is, which registers are used in memory addressing.

The effective address is computed as *Effective Address = Base + (Index*Scale) + Displacement*, where *Base* and *Index* are values stored in any general purpose register, *Scale* is a value of 1,2,4 or 8 and *Displacement* is an 8-bit, 16-bit, or 32-bit value encoded as part of the instruction. For example, the instruction **mov [edi+ecx] + 0x10, eax** will write to the memory at the location edi+ecx+0x10.

To learn the effective address, we assume that the zero page is not allocated and can be used to trigger page faults. Then, to identify whether the instruction is accessing memory, we follow the approach in Algorithm 1. Of importance are the steps taken in lines 7-12 where the hypervisor traps the page fault, inspects the fault code and infers whether the unknown instruction attempted to read or write memory.

To determine the specific registers and the displacement value used to address memory we solve a system of linear equations for two unknowns using unique prime values (acquired in lines 6 and 9). The added capability of memory access identification improves our ability to unveil instructions by over 30%.

Algorithm 1 Identify memory access type (*i.e.,* read or write) and the specific register being accessed

```
 1: Save the current execution state
 2: Enable guest page fault interception
 3: Set registers to unique prime values
 4: Allow guest to re-execute the instruction
 5: if Fault Type == Page Fault then
 6:     Save accessed memory address
 7:     Set registers to unique prime values
 8:     Allow guest to re-execute the instruction
 9:     Save accessed memory address
10: else
11:     No memory access
12: end if
13: Restore the saved execution state
```

4.3 Building application fingerprints

In our approach, we use data collected via Instruction Based Sampling (IBS) [13] to identify the applications running within an encrypted VM. Instruction Based Sampling was introduced to provide detailed application performance information. IBS provides sampling information, collected once every t instructions. After an instruction is sampled, the information is stored in a set of model specific registers and a non-maskable interrupt is raised to indicate the availability of the instruction data. IBS offers two modes of operation: tracking instruction fetches (coined *fetch sampling*) and instruction execution performance (coined *op sampling*). Samples collected via fetch sampling detail performance of an instruction fetch whereas samples collected in the op sampling mode provide information on the retired instructions including the virtual address of the retired instruction, the type of instruction (*e.g.,* branch, load, store), the virtual and physical addresses of accessed memory, the virtual address of the branch target, the type of the branch and the result of branch prediction. Note that we use the information on retired instructions because the data collected in fetch sampling mode is speculative (*i.e.,* the samples may represent instructions that were executed but not retired).

While op sampling may seem well suited for our goals, there are several limitations with using IBS. For one, the collection of the sample and the notification of its availability are asynchronous, resulting in skid between the time the measurement was taken and the time the sample is made available, thus decreasing the maximum sampling frequency. Additionally, IBS samples have no indicator that helps distinguish whether they originate from the kernel or from userspace. Alas, we must find a way to pinpoint the source of the sampled events (*i.e.,* kernel versus userspace) in order to isolate the process from which the sample was drawn. Later on, we address how we overcome these hurdles. Before doing so, we describe our approach for IBS-based application fingerprinting.

4.3.1 *IBS-based fingerprinting.*

The data obtained using IBS op sampling mode must be preprocessed to separate events from different sources. In what follows, we assume that the entity controlling the hypervisor (*i.e.,* cloud provider or a malicious tenant that compromised it) has knowledge of the *host* operating system. Given that assumption, we can safely discard all samples from the known address range of the host kernel. All the remaining samples then belong to guest VMs. Knowing, for example, that in Linux the OS kernel is always mapped in the upper 48 bits of the address space, samples from the guest kernel can also be discarded. Additionally, based on the knowledge that binaries and shared libraries are mapped to distinct memory ranges, the remaining samples can be further separated. Specifically, we cluster the samples into 32 MB bins,[4] starting from the lowest observed virtual address. Given each bin, we select the samples containing return instructions and create an ordered list of their virtual addresses $va_0 \dots va_k$. From that list we generate a fingerprint of the unknown application, u, as a vector of distances $\vec{u} = (d_1 \dots d_k)$ where $d_i = va_i - va_1$. Essentially, this gives us a peek into the binary layout of an unknown application. Said another way, we measure distances from some function boundary early on in an application to other function boundaries in later parts of that application. Note that k varies based on the number of return instructions observed during the period we collect IBS data.

Algorithm 2 Matching IBS based fingerprints

```
1: for all signatures $\vec{r}$ in $\mathcal{R}$ do
2:     for all $r_i \in (r_1 \dots r_m)$ do
3:         if All distances in $\vec{u}$ match sums of consecutive distances starting from $r_i$
   then
4:             Fingerprint $\vec{u}$ matches application $\vec{r}$
5:         end if
6:     end for
7: end for
```

4.3.2 *Application reference set.*

An application reference describes the layout of all functions in a given binary. Specifically, we leverage the distance between function returns to describe the layout. The reference is a vector $\vec{r} = (r_1, r_2, \dots, r_m)$ of distances between all the return instructions in a binary. The size, m, of the reference depends on the number of functions in an application and can range from a few up to tens of thousands.

4.3.3 *Fingerprint matching.*

Fingerprint matching proceeds as one would expect: we identify the unknown image by the sequence of distances between the observed return instructions collected using IBS compared with an off-line reference of target applications curated using binary disassembly. We denote that datastore containing the full fingerprints for all applications of interests (*i.e.,* $[\vec{r_1}, \dots, \vec{r_N}]$ as $\mathcal{R}$. Essentially, $\vec{u}$ is a binary-level fingerprint that captures a small fragment of the unknown application's structure. The fingerprint in Figure 6 (in the Appendix) consists of two distances computed from a set of three addresses of return instructions in the IBS data.

Inputs: (i) A reference database $\mathcal{R}$ of application layouts. Each layout in $\mathcal{R}$ is a vector of distances between successive returns. (ii) A vector $\vec{u}$ of distances between the first seen return instruction and all the other return instructions within a bin. The crux of the Algorithm 2 lies in line 3, where a comparison is made between consecutive distances in $\vec{r}$ and observed distances in $\vec{u}$. The search starts with the first distance, $r_{i=1}$ from $\vec{r}$ then adds consecutive distances from $\vec{r}$ to test if they can match distances in $\vec{u}$. If any of the distances from $\vec{u}$ cannot be matched to a sum of distances from starting position r_i, the search restarts at r_{i+1}. The unknown vector is considered identified if, and only if, all the distances from $\vec{u}$ are matched to sums of distances from $\vec{r}$. The example presented in Figure 6 is a positive match: distance d_1 is equal to the sum of $r_3 + r_4 + r_5 + r_6$ and d_2 is equal to r_6.

[4]Value derived from an empirical evaluation of the average size of Ubuntu binaries.

5 EV

To valid
form wh
can effic
tions wi
data bei
within t
style att
target V
our atte

Exper

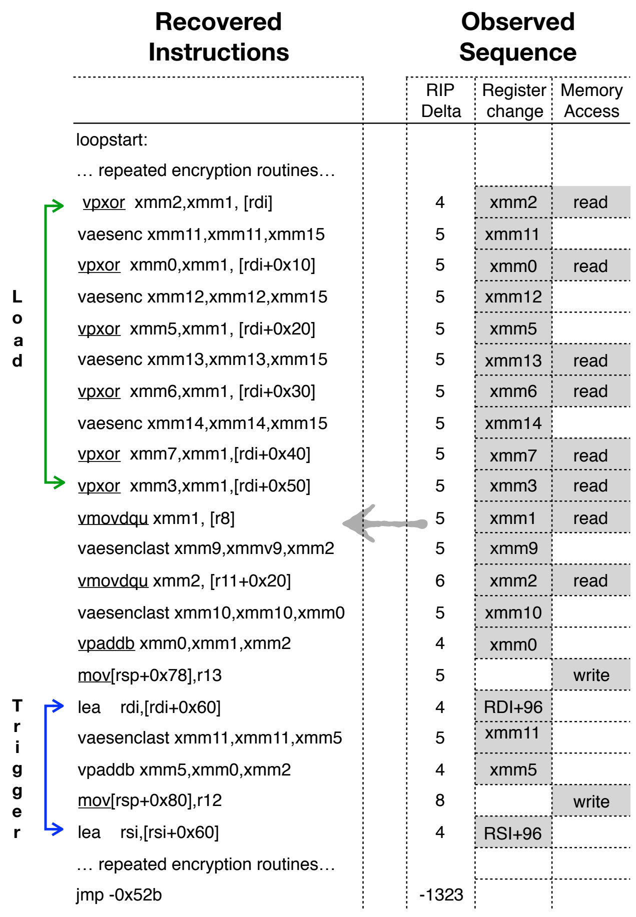
All expe
ware. W
Epyc 73
Ubuntu
kernel a
Guest V
and run
introspe
KVM ke

5.1 A

To demo
a SEV-en
HTTPS.
network
commun
the arch
istics of
to extra

In Nginx, an initialization process creates new sockets (`socket()`) and binds to HTTP/HTTPS ports (`bind()`, `listen()`). After setting up the socket, the initialization process clones itself (`clone()`) to create the master process. The master process then clones (`clone()`) itself to create worker processes that handle incoming connections. The worker process waits for the incoming connections (`epoll()`), then accepts an incoming connection (`accept()`), transfers data to the client (`recvfrom()`, `writev()`), and finally terminates the connection (`close()`) and returns to the waiting state. We hyper-step in between the `recvfrom()` and `writev()` system calls to recover the data processed by the server. A pictorial representation is shown in Figure 5 in Appendix C. In TLS secured connections, the worker process conducts a handshake [43] and starts transmitting data when it is complete. The instruction instrumentation is performed only on the data exchange part of communication to reduce the processing overhead. The exchanged data is decrypted and encrypted using the hardware AES engine that developers can utilize via the AES-NI instruction set extension. Thus, observation of the XMM registers allows us to extract the plaintext of the request and the response as well as the AES keys. In the experiment, the server used the Diffie-Hellman key exchange protocol and encrypted the traffic using 128-bit AES in Galois Counter mode. (OpenSSL cipher suite 0x9e: DHE-RSA-AES128-GCM-SHA256.)

[5] Available at https://github.com/AMDESE/AMDSEV/.



	RIP Delta	Register change	Memory Access
loopstart:			
… repeated encryption routines…			
vpxor xmm2,xmm1, [rdi]	4	xmm2	read
vaesenc xmm11,xmm11,xmm15	5	xmm11	
vpxor xmm0,xmm1, [rdi+0x10]	5	xmm0	read
vaesenc xmm12,xmm12,xmm15	5	xmm12	
vpxor xmm5,xmm1, [rdi+0x20]	5	xmm5	
vaesenc xmm13,xmm13,xmm15	5	xmm13	read
vpxor xmm6,xmm1, [rdi+0x30]	5	xmm6	read
vaesenc xmm14,xmm14,xmm15	5	xmm14	
vpxor xmm7,xmm1,[rdi+0x40]	5	xmm7	read
vpxor xmm3,xmm1,[rdi+0x50]	5	xmm3	read
vmovdqu xmm1, [r8]	5	xmm1	read
vaesenclast xmm9,xmmv9,xmm2	5	xmm9	
vmovdqu xmm2, [r11+0x20]	6	xmm2	read
vaesenclast xmm10,xmm10,xmm0	5	xmm10	
vpaddb xmm0,xmm1,xmm2	4	xmm0	
mov[rsp+0x78],r13	5		write
lea rdi,[rdi+0x60]	4	RDI+96	
vaesenclast xmm11,xmm11,xmm5	5	xmm11	
vpaddb xmm5,xmm0,xmm2	4	xmm5	
mov[rsp+0x80],r12	8		write
lea rsi,[rsi+0x60]	4	RSI+96	
… repeated encryption routines…			
jmp -0x52b	-1323		

Figure 3: Abstraction of going from observed register changes to inferred instructions of the encryption loop of the HTTPS response. The underlined instructions are those inferred using the contextual information within the scope of the analysis.

When the hypervisor detects the sequence of system calls that indicate the server is about to receive data from a network socket, we transition to the second stage of the attack. In stage two, the interception of the SSL_read function allows us to unmask the instructions that process the plaintext of the request and the response sent over the TLS protected network connections — all via examination of the general purpose registers. We note that unmasking of the instructions *significantly* simplifies the process of recovering the encrypted data in that it allows us to simply copy the plaintext from the register when the decryption is complete, rather than sift through intermediary values of the encryption process.

5.1.1 *Under the hood.*

Recall that our techniques for recovery of the instructions shown in Figure 3 leverage our instruction identification (§4.2.1) methodology and memory access detection (§4.2.2) technique. For brevity, we skip the bulk of the encryption and focus the reader's attention on the sections responsible for loading data from memory, storing the results, and the loop construct.

Using the memory access tracking, we identify access patterns of the instructions in the trace. Armed with the knowledge of the memory accesses one can easily identify `mov` instructions, where

Table 2: Overhead of the TLS recovery PoC

Size	Baseline (µs)	System call tracking (µs)	Full PoC (µs)
8k	4539	6432	6476
32k	7604	8107	8515
128k	13801	13383	14591

the destination operand is memory, and distinguish instructions that load register contents from memory from those that perform arithmetic or logical operations. Finally, given that our coarse grain tracking was used to trigger hyper-stepping, we can use that knowledge to match the recovered set of instructions as the loop of the `CRYPTO_128_unwrap` function.

With the acquired knowledge of the code layout, we can set a new finer-grained trigger point, Δ' (`lea rdi, [rdi+0x60]`), to the location in the loop where accessing the registers will disclose the plaintext in the XMM registers. The underlined instructions in Figure 3 are those inferred using the contextual information within the scope of the instruction under analysis.

To be sure that we reached the critical section of the code where we can extract the plaintext from the registers, we verify that the trace observed while hyper-stepping contains the sequence: [`RIP+4, RDI+96; RIP+4, new value of XMM11; RIP+4, new value of XMM5; RIP+8, memory write; RIP+4, RSI+96`].[6] **Once verified, we copy the contents of the `XMM` registers and reassemble the HTTPS stream**.

5.1.2 Results. To gain insights into the run time overhead, we averaged the processing time of 25 requests for varying sizes of requested data. The average round trip time for a packet between the client and the server in our setup was 5ms (5000 µs). Our results provided in Table 2 show that the user perceived delay is slightly less than 1*ms per 32 kb* of data. The overhead would be even lower if the adversary only needs to instrument the request to obtain the requested URL, user credentials and any other information that is necessary to reissue the request.

5.2 Attack on SEV: Injecting Keys

Next, we examine how a malicious hypervisor can thwart the full disk encryption and the memory encryption that are used by the guest. To that end we demonstrate how the malicious hypervisor can intercept data from an encrypted hard drive, and inject faux data into the datastream. For pedagogical purposes, we focus on the `read()` system call and how it is used to provide access to various devices. For the remaining discussion, it suffices to know that control flows from the system call entry point through the Virtual File System (VFS) and the extended file system drivers to a file system agnostic function (*i.e.,*`copy_user_generic()`) that is responsible for copying the data between kernel space and user space memory. For all file systems supported in the Linux kernel, we found that this generic function checks the kernel data structure for information on the available CPU extensions and based on that information invokes a specific low level assembly function.

An in-depth analysis of the kernel initialization functions showed that a single invocation of the CPUID instruction is used only to specify the processor features for the purpose of selecting the memory copy instruction. Specifically, the OS can be forced to use less efficient register-to-register copy operations instead of fast string operations by masking certain bits in the results of the CPUID instruction. For example, in the Linux kernel, the decision to use specific memory copy instructions is based on available CPU features.[7] Moreover, the check of the availability of the CPU specific features[8] can be manipulated by spoofing the value returned by the CPUID instruction to force the kernel not to use memory to memory copy (*i.e.,* `rep movs` instructions).

By augmenting the same approach outlined in §5.1, we show how a malicious hypervisor can gain arbitrary user access in the SEV-enabled enclave. The attack we explore is a variation of an Iago attack [8], which is the term for an attack where the response from untrusted kernel undermines the security of the user space process. Rather than modifying the kernel, however, in our case the malicious hypervisor performs a man in the middle attack between the user space and kernel.

We exploit the fact that the target of the attack (*i.e.,* the OpenSSH server) performs a series of sanity checks during the public-key authentication process. Specifically, first it checks whether the `authorized_keys` file exists, and if so, verifies the permissions of the directory structure holding the file. If the checks are satisfied, the contents of the file are read and the authentication process attempts to verify the key. This sequence allows one to build a unique application profile. In our instantiation of the attack, when the trigger (*i.e.,* a sequence of system calls that indicate the reading of the user `authorized_keys`) is detected, the malicious hypervisor executes the man in the middle attack.

Next, we need only hyper-step the `copy_user_generic()` routine in the kernel. First, the contents of the kernel buffer are copied to the userspace buffer. Next, when the copy is complete (*i.e.,* when the counter value in register `ECX` reaches 0), we artificially increase the amount of data to be copied. We then feed the faux key to the user space buffer by modifying the data in the source register. Finally, the return value of the system call (stored in a register) is adjusted to reflect the new length of the data.

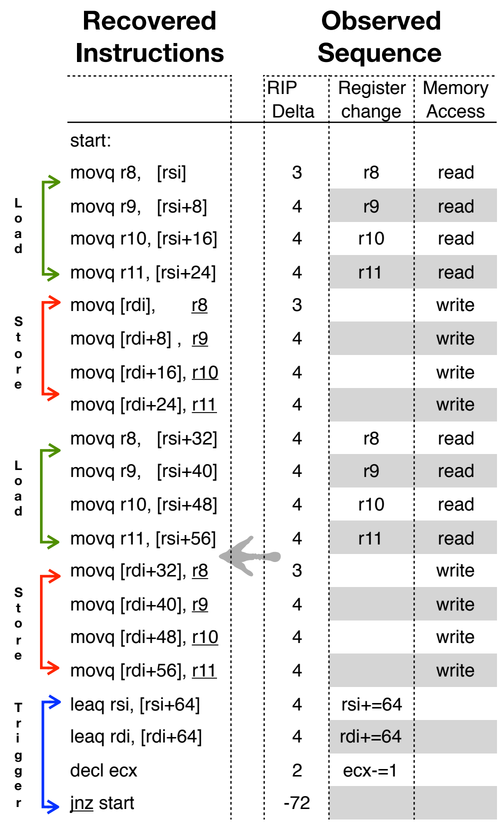
5.2.1 Under the hood. Similar to the TLS proof of concept, the change in `RIP`, the register changes, and the type of memory accesses are all used to unmask the sequence of instructions. In Figure 4, the order of the registers used in the store section (red arrow) is guessed based on the order of the instructions in the load section (green arrow). That is, we assume the information is written in the same order as it was read. In the case of the inferred `jnz` we know from the change of the instruction pointer that the instruction is a jump, because of the negative change of `RIP`. Additionally, the previously decoded instructions indicate a decrement of the counter register `RCX`, and so we surmise that the jump is of the form "jump if zero flag is not set" (*i.e.,* `jnz`). The other instructions and operands are exactly unmasked from the observed change in `RIP`, the identification of the memory access type, and the determination of what register was accessed.

However, unlike in the previous case, the initial cost of finding the fine-grained trigger is significantly less. Based on the semantics

[6]Static analysis of the Nginx binary and the 35 shared libraries it loads revealed that the sequence we use is unique.

[7] https://elixir.bootlin.com/linux/v4.15/source/arch/x86/include/asm/uaccess_64.h#L36

[8]See https://elixir.bootlin.com/linux/v4.15/source/arch/x86/kernel/cpu/amd.c#L623



	Recovered Instructions	Observed Sequence		
		RIP Delta	Register change	Memory Access
	start:			
Load	movq r8, [rsi]	3	r8	read
Load	movq r9, [rsi+8]	4	r9	read
Load	movq r10, [rsi+16]	4	r10	read
Load	movq r11, [rsi+24]	4	r11	read
Store	movq [rdi], r8	3		write
Store	movq [rdi+8] , r9	4		write
Store	movq [rdi+16], r10	4		write
Store	movq [rdi+24], r11	4		write
Load	movq r8, [rsi+32]	4	r8	read
Load	movq r9, [rsi+40]	4	r9	read
Load	movq r10, [rsi+48]	4	r10	read
Load	movq r11, [rsi+56]	4	r11	read
Store	movq [rdi+32], r8	3		write
Store	movq [rdi+40], r9	4		write
Store	movq [rdi+48], r10	4		write
Store	movq [rdi+56], r11	4		write
Trigger	leaq rsi, [rsi+64]	4	rsi+=64	
Trigger	leaq rdi, [rdi+64]	4	rdi+=64	
Trigger	decl ecx	2	ecx-=1	
Trigger	jnz start	-72		

Figure 4: Abstraction of going from observed register changes to unmasked instructions. The underlined instructions are inferred based on contextual information within the scope of the analysis.

of the observed system call, the target code section, and the information available when the trigger is detected, we can limit the amount of hyper-steps required to reach the critical instructions. Specifically, when the trigger (*i.e.,* read system call) is detected, arguments of that system call including the pointer to the destination buffer are visible to the introspection mechanism. We set a hardware breakpoint on the destination buffer pointed to by the second argument of the `SYS_read` system call. This allows us to avoid hyper-stepping through the kernel virtual file system function stack and instead start the introspection inside the `copy_user_generic()` function.

To be sure that we can now safely extend the loop without unwanted side effects in the guest, we assure that in the observed trace there is a change of register state matching the tuple `[RIP+4,RSI+64; RIP+4,RDI+64; RIP+2,RCX -1]` and `RCX` has reached zero. Once satisfied, we inject our faux key.

5.2.2 Results. For the attack, we inject a 2048-bit RSA public key belonging to the adversary. The key length (512 bytes) mandates that we complete 8 iterations of the loop (in Figure 4) to inject the key into the SEV-protected guest. Pulling off the attack requires a mere 160 hyper-steps, which is imperceptible during the SSH session establishment process. **Henceforth, the adversary is free to execute any code within the VM and poke around at will.**

5.3 Attack on SEV-ES: Application Fingerprinting

To show that we can successfully identify applications running in a SEV-ES protected enclave, we performed an empirical evaluation using Cloudsuite [39]. Cloudsuite offers a series of benchmarks for cloud services, and uses real world software stacks under realistic workloads. The web serving benchmark consists of a web server (Nginx) with the accompanying infrastructure (*i.e.,* PHP, MySQL, Memcached) serving an open source social network engine called Elgg. In our setup, the benchmark is hosted in a virtual machine running Ubuntu.

Our reference datastore, $\mathcal{R}$, consists of binary fingerprints generated from the disassembly of all the system binaries of Ubuntu (*i.e.,* /bin/ /sbin /usr/sbin /usr/bin /systemd). $\mathcal{R}$ consists of 1465 entries. At runtime, we generate partial fingerprints for unknown applications using a custom tool based on the AMD IBS Research Toolkit [3]. The modifications were done to reduce the number of samples collected from the host OS and to restrict data collection to the CPU core running the guest VM. We are able to sample *once every eight hundred instructions* due to the skid in IBS.

To get a sense of the diversity of the layouts in $\mathcal{R}$, for all pairs i, j, we examine the matching subsequences in $\vec{r_i}$ and $\vec{r_j}$. We say that two applications have identical layouts if they have the same number of functions $|\vec{r_i}| = |\vec{r_j}|$, and the lengths of the respective functions are equal. Our analyses show that the length of matching subsequences is indeed a strong indicator of binary similarity; for example, the longest matching sequence for distinct applications in $\mathcal{R}$ had only six elements. Obviously, the load on the server and the duration of the observation period have direct impact on the quality of the fingerprints we collect at runtime. Intuitively, an idle application will generate a limited amount of performance data. Additionally, given the sampling frequency limitations of IBS (see §4.3), there is no way to guarantee that the observed IBS data will contain return instructions. Thus, the longer $\vec{u}$ is, the more confidence we can have in knowing whether it matches one of the target applications in $\mathcal{R}$.

5.3.1 Results. To that end, the load of the VM was varied by issuing varying number of login requests to the Elgg community site running in the targeted VM. Our results show that in cases in which we collected fingerprints comprising more than three distances, we can successfully identify all the applications belonging to the Cloudsuite web serving benchmark (*i.e.,* Nginx, PHP, MySQL and memcached) as well as other system applications (*e.g.,* systemd, snapd). Table 3 presents the relationship between the average number of return instruction given varying number of web requests, as well as the true and false positive rates. At $|\vec{u}| > 3$, we attain a TP rate of 1 and FP rate of 0.000006. If we allow the adversary to collect more data, the FP drops to 0.0 once $|\vec{u}| > 6$. Even at that threshold, the overhead is negligible. To measure overhead, we averaged the processing time of 100 login requests. We observed an average overhead of 30 µs, which is imperceptible to an end-user.

More interestingly, we find that this binary fingerprinting technique can even distinguish between applications and compiler versions. To demonstrate that, we extended $\mathcal{R}$ to include the disassembly of 10 different versions of Nginx compiled using two versions of GCC. For example, two recent versions of Nginx (v1.15.8 and

Table 3: Application Identification Success Rate

# of HTTP requests	Avg. #of return instr.	TP rate	FP rate ($\lvert\vec{u}\rvert > 3$)
5	3.78	1	$7.1 * 10^{-6}$
10	6.91	1	$1.42 * 10^{-5}$
25	14.06	1	0
50	23.48	1	0

v1.14.2) compiled using the same compiler (GCC version 7.3) shared a sequence of 108 distances. On the other hand, the same version of Nginx compiled using two different versions of GCC (v7.3 vs v5.4) had no shared sequences. Our evaluation shows that given fingerprints longer than four distances we are able to distinguish the exact version of an application. As long as we collect one return instruction that falls outside of the matching sequence of two versions of the same application, we can distinguish between them.

The ability to precisely identify software running within an encrypted VM has far reaching implications. First, an honest cloud provider can use application fingerprinting to identify potentially unwanted software and violations of acceptable use policy. On the other hand, a malicious adversary performing reconnaissance using the IBS-based inference attack gains valuable insight that can be leveraged for further attacks, *e.g.,* ROP. Identifying the specific version of an application has the advantage that it allows an adversary to target specific vulnerabilities. Third, the IBS data can be used to undermine user space Address Space Layout Randomization (ASLR). Recall that ASLR randomizes module base addresses (*i.e.,* the address at which the application is loaded in memory). Since $\vec{u}$ is built using the virtual addresses of return instructions, once $\vec{u}$ is matched to some $\vec{r}$, the adversary can use that knowledge to compute the base address of r — thereby defeating ASLR.

6 DISCUSSION AND POTENTIAL MITIGATIONS

Although AMD is the first vendor to provide a commodity solution for transparently encrypting guest memory, there is a large body of work that attempts to protect the confidentiality and integrity of application data even in the event of OS or hypervisor compromise [5, 9, 11, 15, 18, 25, 47, 48, 50, 51, 55, 56]. Henson and Taylor [23] provide a systematic assessment of many of these approaches. Pertinent to this work are the ideas in Overshadow [9], where isolation capabilities of the virtualization layer are extended to allow protection of entities inside of a virtual machine via a technique called *cloaking*. A similar idea was also proposed by Xia et al. [56], but with the touted advantage of having a smaller trusted computing base for their shim. Our work demonstrates in a definitive way that the access to general purpose registers and an ability to interrupt the guest, are sufficient to *unveil executing instructions and recover data* that is otherwise stored in an encrypted memory and storage. *None* of these works take into account protection against this new class of inference attacks presented herein.

Unfortunately, while SEV-ES prevents the hypervisor from inspecting and modifying general purpose registers, virtualization support for this extension has only just become available.[9] Until the support for SEV-ES matures, we offer an interim solution that limits the ability of the hypervisor to force automatic exits as a way to mitigate the register inference attacks. The hypervisor should never be allowed to intercept any events that are under the control of the guest. But, this is no easy feat, as there is an extensive list [2, §15.9-15.12] of intercepts and traps, many of which are supported for legacy reasons (*e.g.,* access to control register 3 that was used in shadow page table implementations), debugging functions, or obscure functionality. Nevertheless, we suggest the use of trap and interrupt masks that are applied by the processor to the trap and interrupt vectors saved in the virtual machine control block. During the transition from the hypervisor to the guest using the `VMRUN` instruction, the processor should raise the general protection fault if the intercept and trap controls in the VMCB do not conform to the allowed masks. The masks and the change of the `VMRUN` instruction could be delivered in the form of a microcode update for the main CPU, similarly to the way microcode patches were distributed to mitigate the Spectre and Meltdown vulnerabilities [1].

[9]A modified Linux kernel is available at: https://github.com/AMDESE/linux/commits/sev-es-4.19-v2. The code enabling SEV-ES was made available on May 17, 2019

Per our structural inference attack on SEV-ES, the knee-jerk reaction might be to disable the IBS subsystem. However, it is possible to use software workarounds [3] to enable IBS in software. Worse, it is not possible for the guest to determine whether IBS is enabled or not, since the hypervisor ultimately controls the Model Specifics Registers used to program the IBS subsystem. Moving forward, to prevent the application fingerprinting attack, we suggest that the performance measurement subsystem differentiate the data collected from the guest and the host and discard the samples from the guest when secure encrypted virtualization is enabled.

7 CONCLUSION

To address cloud confidentiality, virtualization technologies have recently offered encrypted virtualization features that support transparent encryption of memory as a means of protection against malicious tenants or even untrusted hypervisors. In this paper, we examine the extent to which these technologies meet their goals. In particular, we introduce a new class of inference attacks and show how these attacks can breach the privacy of tenants relying on secure encrypted virtualization technologies. As a concrete case in point, we show how the security of the Secure Encrypted Virtualization (SEV) platform can be undermined. Additionally, we show that even when additional state is encrypted (*e.g.,* as proposed under the SEV-ES extension where the state of general purpose registers is also encrypted), an adversary may still mount application fingerprinting attacks, rendering those protections less effective than first thought. We provide suggestions for mitigating the threat posed by some of these attacks in the short term.

8 ACKNOWLEDGMENTS

We thank Richard Strong and Caleb Loverro for fruitful discussions and feedback on an earlier draft of this paper. We also thank David Kaplan and Joe Greathouse for their feedback and insights regarding IBS. We express our gratitude to Murray Anderegg for his assistance with deploying the infrastructure used in this study. This work was supported in part by the Department of Defense (DoD) under award FA8750-17-C-0016, the Office of Naval Research (ONR) under award N00014-17-1-2891, the National Science Foundation (NSF) under award CNS-1749895, and the Defense Advanced Research Projects Agency (DARPA) under award D18AP00045. Any opinions, findings, and conclusions expressed herein are those of the authors and do not necessarily reflect the views of the DoD, ONR, NSF, or DARPA.

REFERENCES

[1] L. Abrams. Intel releases linux CPU microcodes to fix meltdown and spectre bugs, 2017.

[2] AMD. AMD64 architecture programmer's manual volume 2: System programming. http://support.amd.com/TechDocs/24593.pdf, 2017.

[3] AMD. AMD research instruction based sampling toolkit. https://github.com/jlgreathouse/AMD_IBS_Toolkit, 2018.

[4] AMD. SEV-ES guest-hypervisor communication block standardization. https://developer.amd.com/wp-content/resources/56421.pdf, 2019.

[5] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 267–283, 2014.

[6] A. K. Biswas, D. Ghosal, and S. Nagaraja. A survey of timing channels and countermeasures. *ACM Computer Surveys*, 50(1):6:1–6:39, Mar. 2017.

[7] J. V. Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *USENIX Security Symposium*, pages 1041–1056, 2017.

[8] S. Checkoway and H. Shacham. Iago attacks: why the system call API is a bad untrusted RPC interface. In *Architectural Support for Programming Languages and Operating Systems*, pages 253–264, 2013.

[9] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. *SIGPLAN Not.*, 43(3): 2–13, Mar. 2008.

[10] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, pages 857–874, 2016.

[11] J. Criswell, N. Dautenhahn, and V. Adve. Virtual ghost: Protecting applications from hostile operating systems. In *ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 81–96, 2014.

[12] CTS-LABS. Severe security advisory on AMD processors. https://amdflaws.com, 2018.

[13] P. J. Drongowski. Instruction-based sampling: A new performance analysis technique for AMD family 10h processors. https://developer.amd.com/wordpress/media/2012/10/AMD_IBS_paper_EN.pdf, 2007.

[14] Z.-H. Du, Z. Ying, Z. Ma, Y. Mai, P. Wang, J. Liu, and J. Fang. Secure Encrypted Virtualization is Unsecure https://arxiv.org/pdf/1712.05090.pdf, 2017.

[15] R. Elbaz, D. Champagne, C. Gebotys, R. B. Lee, N. Potlapally, and L. Torres. Hardware mechanisms for memory authentication: A survey of existing techniques and engines. In *Transactions on Computational Science IV*, pages 1–22, 2009.

[16] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for Unix processes. In *IEEE Symposium on Security & Privacy*, 1996.

[17] J. Greene. Intel Trusted Execution Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/trusted-execution-technology/trusted-execution-technology-security-paper.html, 2012.

[18] S. Gueron. Memory encryption for general-purpose processors. *IEEE Security Privacy*, 14(6):54–62, Nov 2016.

[19] S. Gueron. A memory encryption engine suitable for general purpose processors. ePrint Archive, Report 2016/204, 2016. https://eprint.iacr.org/2016/204.

[20] D. Gullasch, E. Bangerter, and S. Krenn. Cache games – bringing access-based cache attacks on AES to practice. In *IEEE Symposium on Security & Privacy*, pages 490–505, May 2011.

[21] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *USENIX Security Symposium*, pages 45–60, 2009.

[22] Y. Hebbal, S. Laniepce, and J. M. Menaud. Virtual machine introspection: Techniques and applications. In *International Conference on Availability, Reliability and Security*, pages 676–685, Aug 2015.

[23] M. Henson and S. Taylor. Memory encryption: A survey of existing techniques. *ACM Computer Survey*, 46(4):53:1–53:26, Mar. 2014.

[24] F. Hetzelt and R. Buhren. Security analysis of encrypted virtual machines. In *ACM International Conference on Virtual Execution Environments*, pages 129–142, 2017.

[25] S. Jin, J. Ahn, S. Cha, and J. Huh. Architectural support for secure virtualization under a vulnerable hypervisor. In *International Symposium on Microarchitecture*, pages 272–283, Dec 2011.

[26] D. Kaplan. Protecting VM register state with SEV-ES. http://support.amd.com/TechDocs/Protecting%20VM%20Register%20State%20with%20SEV-ES.pdf, 2017.

[27] D. Kaplan, J. Powell, and T. Woller. AMD memory encryption. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf, 2016.

[28] Y. Kim, R. Daly, J. Kim, C. Fallin, J. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Rowhammer: Reliability analysis and security implications. *CoRR*, abs/1603.00747, 2016.

[29] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. https://arxiv.org/abs/1801.01203, 2018.

[30] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *USENIX Security Symposium*, pages 523–539, 2017.

[31] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX Security Symposium*, pages 557–574, 2017.

[32] T. Lendacky. [RFC PATCH v1 00/18] x86: Secure memory encryption (AMD). https://www.mail-archive.com/linux-doc@vger.kernel.org/msg02713.html, 2016.

[33] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. https://arxiv.org/abs/1801.01207, 2018.

[34] Y. Lyu and P. Mishra. A survey of side-channel attacks on caches and countermeasures. *Journal of Hardware and Systems Security*, Nov 2017.

[35] G. Maisuradze and C. Rossow. Speculose: Analyzing the security implications of speculative execution in CPUs. https://arxiv.org/abs/1801.04084, 2018.

[36] M. Morbitzer, M. Huber, J. Horsch, and S. Wessel. Severed: Subverting amd's virtual machine encryption. In *European Workshop on System Security*, 2018.

[37] M. Morbitzer, M. Huber, and J. Horsch. Extracting secrets from encrypted virtual machines. In *ACM CODASPY*, 2019.

[38] National Security Institute. *Department of Defense Trusted Computer System Evaluation Criteria*. Department of Defense, 1985.

[39] E. PARSA. Cloudsuite. http://cloudsuite.ch/, 2018.

[40] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard. DRAMA: Exploiting DRAM addressing for cross-cpu attacks. In *USENIX Security Symposium*, pages 565–581, 2016.

[41] J. V. B. F. Piessens and R. Strackx. Sgx-step: A practical attack framework for precise enclave execution control. In *Workshop on System Software for Trusted Execution (SysTEX)*, 2017.

[42] N. A. Quynh. Operating system fingerprinting for virtual machines. Defcon, 2010.

[43] E. Rescorla. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley, 2001.

[44] J. Sharkey. Breaking hardware-enforced security with hypervisors. Black Hat USA, 2016.

[45] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing page faults from telling your secrets. In *ACM Conference on Computer and Communications Security*, pages 317–328, 2016.

[46] R. Spreitzer, V. Moonsamy, T. Korak, and S. Mangard. SoK: systematic classification of side-channel attacks on mobile devices. *CoRR*, 2016.

[47] J. Szefer and R. B. Lee. Architectural support for hypervisor-secure virtualization. *SIGPLAN Not.*, 47(4):437–450, Mar. 2012.

[48] B.-P. Tine and S. Yalamanchili. Pagevault: Securing off-chip memory using page-based authentication. In *Proceedings of the International Symposium on Memory Systems*, pages 293–304, 2017.

[49] Trusted Computing Group. TPM Main: Part 1 – Design Principles. https://trustedcomputinggroup.org/resource/tpm-main-specification/, 2003.

[50] T. Unterluggauer, M. Werner, and S. Mangard. Securing memory encryption and authentication against side-channel attacks using unprotected primitives. *IACR ePrint Archive*, 2017:663, 2017.

[51] A. Vahldiek-Oberwagner, E. Elnikety, D. Garg, and P. Druschel. Erim: Secure and efficient in-process isolation with memory protection keys. https://arxiv.org/abs/1801.06822, 2018.

[52] VMware. Securing the cloud: A review of cloud computing, security implictions, and best practices, 2009.

[53] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *ACM Conference on Computer and Communications Security*, pages 2421–2434, 2017.

[54] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: alternative data models. In *IEEE Symposium on Security & Privacy*, pages 133–145, 1999.

[55] M. Werner, T. Unterluggauer, R. Schilling, D. Schaffenrath, and S. Mangard. Transparent memory encryption and authentication. In *Field Programmable Logic and Applications (FPL)*, pages 1–6, 2017.

[56] Y. Xia, Y. Liu, and H. Chen. Architecture support for guest-transparent vm protection from untrusted hypervisor and physical attacks. In *International Symposium on High Performance Computer Architecture*, pages 246–257, Feb 2013.

[57] Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu. One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation. In *USENIX Security Symposium*, pages 19–35, 2016.

[58] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security & Privacy*, pages 640–656, 2015.

A ETHICAL CONSIDERATIONS

As is common in the computer security landscape, there is an intricate dance between defensive and offensive research. We have shared our results with AMD regarding the power of inference attacks, and have incorporated some of the feedback into the paper. This new class of attacks is a direct outcome of not having the ability to inspect main memory.

B INTROSPECTION ALGORITHM

The procedure we use to selectively hyper-step is presented in Algorithm 3.

Algorithm 3 Introspection using Trigger Points

```
1:  Off-line: identify critical code section, generate profile, set candidate trigger $\Delta$
2:  loop Introspection
3:      Identify target (§4.1) using profile
4:      if trigger point $\Delta$ reached then
5:          repeat
6:              Hyper-step (§4.2) the target
7:              Unveil likely instructions (§4.2.1)
8:              Locate fine-grained trigger $\Delta$'
9:              if $\Delta$' found then
10:                 Set $\Delta$ = $\Delta$'
11:             end if
12:             Exfiltrate data
13:         until system call invocation
14:     end if
15: end loop
```

C NGINX PROCESS CONTROL

The procedure involved in spawning processes in Nginx is shown in Figure 5. The sequence of system calls spanning init, master, and workers processes (observable in the context of the SEV register

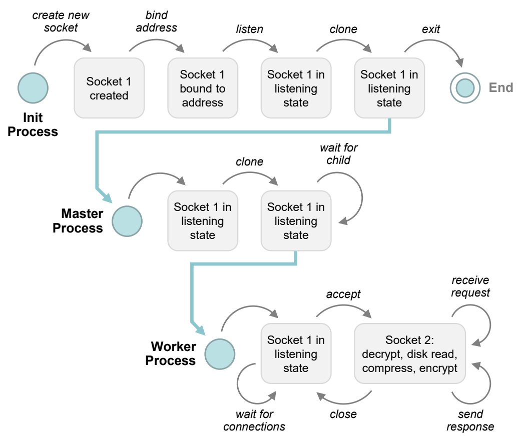
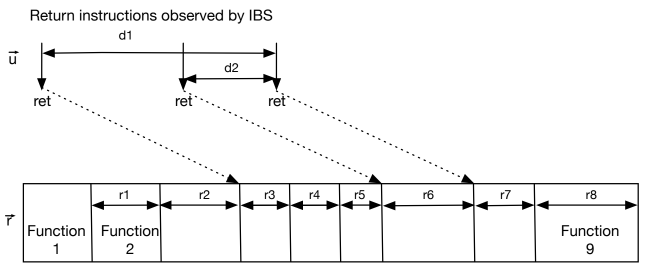

create new socket
bind address
listen
clone
exit
Socket 1 created
Socket 1 bound to address
Socket 1 in listening state
Socket 1 in listening state
End
Init Process
clone
wait for child
Master Process
Socket 1 in listening state
Socket 1 in listening state
accept
receive request
Worker Process
Socket 1 in listening state
Socket 2: decrypt, disk read, compress, encrypt
wait for connections
close
send response

Figure 5: Process control in Nginx.

D IBS BASED FINGERPRINT

In the example presented in Figure 6, the reference consists of eight distances for the nine functions in the application binary image.



Layout of an application obtained via disassembly. Each vertical line represents a return instruction.

Figure 6: Application reference and IBS based fingerprints