Understanding and Exploiting Design Flaws of AMD Secure Encrypted Virtualization

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Mengyuan Li,

Graduate Program in Computer Science and Engineering

The Ohio State University

2022

Dissertation Committee:

Yinqian Zhang, Advisor

Radu Teodorescu, Co-advisor

Feng Qin

Zhiqiang Lin

Abstract

Trusted Execution Environment (TEE) is a blooming direction in the cloud industry. Aiming at protecting cloud user's data in runtime, TEE can enable a lot of new and foreseeable cloud use cases. While enclave-based TEEs such as Intel SGX suffer from a large effort of rewriting existing code, VM-based TEEs such as Intel Trust Domain Extension (TDX) and AMD Secure Encrypted Virtualization (SEV) attract more and more people's attention. Among those VM-based TEEs, AMD SEV is a security extension for the AMD Virtualization (AMD-V) architecture, which is AMD's ambitious movement towards confidential cloud computing. SEV allows one physical server to efficiently run multiple guest virtual machines (VM) concurrently on encrypted memory with the goal of protecting the security of guest VMs even in the presence of a malicious hypervisor. SEV is also believed to be the first and the only commercial VM-based TEE that has already been adopted in Google Cloud and Microsoft Azure at the time of writing. However, the strong assumption of SEV causes uncertainty in its security guarantee. The lack of a systematic security study in this new assumption makes some unexploited vulnerabilities possible. Thus, it is very urgent to fully study SEV's design and help the community better understand SEV.

In this dissertation, we systematically study the structure of AMD SEV's design, including angles from both hardware and software. By comprehensively exploring SEV's different components, we reveal how SEV's hardware and software work together to provide a trusted execution environment, and we also explore several unexploited vulnerabilities in SEV.

Here, we briefly outline five categories of vulnerable designs in SEV and the corresponding security attacks.

In Chapter 3, we exploit the unprotected I/O operations of SEV-enabled VMs and show that the malicious hypervisor can breach the confidentiality and the integrity of guest VMs with the help of these I/O operations.

In Chapter 4, we explore the improper Address Space Identifier (ASID)-based memory isolation and access control. We show that the untrusted hypervisor has control over the VM's ASID without necessary hardware limits. We exploit this design and propose a series of attacks called CROSSLINE attacks. We show that this vulnerable design can be used to decrypt VM's encrypted memory or to momentarily execute arbitrary instructions of the victim VM.

In Chapter 5, we provide the first exploration of TLB management in SEV. We first demystify how SEV extends the TLB implementation and show that the TLB management is no longer secure under SEV's threat model, which allows the hypervisor to poison TLB entries between two processes of a SEV VM. We then present TLB Poisoning Attacks, a class of attacks that break the integrity and confidentiality of the SEV VM by poisoning its TLB entries.

In Chapter 6, we explore the context switch between the guest VM and the host. We show that during context switch, encrypting the virtual CPU's register stored in the VM Save Area is not enough, which allows the privileged adversary to infer the guest VM's execution states or recover certain plaintext. To demonstrate the severity of the vulnerability, we present the CIPHERLEAKS attack, which exploits the ciphertext side channel to steal private keys from the constant-time implementation of RSA and ECDSA in the latest OpenSSL library.

In Chapter 7, we perform a comprehensive study of the ciphertext side channels that widely exist in almost all VM-based TEEs. Our work suggests that the deterministic encryption adopted by almost all commercial VM-based TEEs may potentially leak key-related secrets in most up-to-date mainstream cryptography libraries. The proposed generic ciphertext side-channel attack may exploit the ciphertext leakage from any memory page, including those pages used for kernel data structures, stacks, and heaps.

This dissertation shows that the strong assumption of SEV indeed causes some unexploited vulnerabilities. We have reported our findings to the AMD SEV team and have received very positive feedback (several CVEs, two microcode patches, two cryptography library patches, three official security bulletins and an official white paper for software mitigation[1]). We believe that our work can not only help improve the security of AMD SEV but also help the entire TEE community to build and develop more secure products for all cloud customers.

[1] `https://www.amd.com/system/files/documents/221404394-a_security_wp_final.pdf`

Acknowledgments

It has been a while since I left my hometown, kept pursuing more knowledge, and tried to realize my self-value. Thus, I would like to express my most sincere gratitude here to those who helped and encouraged me during my Ph.D. Period.

I would like to first express my most respectful thanks to my advisor Prof.Yinqian Zhang. He is the best guide I could ever imagine on the road to research. Without him, I would not have been able to begin my Ph.D. program and learn how to become an independent researcher. His patience and professional academic knowledge helped me grow from a freshman to a researcher with certain independent research capabilities. His rigorous scientific attitude, insight into new fields, and curiosity about new things will play a decisive guiding role in my future academic career.

I am also very fortunate to have close cooperation with Prof. Radu Teodorescu, Prof. Srinivasan Parthasarathy, and Prof. Zhiqiang Lin. Prof. Radu Teodorescu's abundant knowledge in architecture and system always gives me different inspiration and guidance. For Prof. Srinivasan Parthasarathy, I am very honored to have three course works with him. His teaching style impressed me a lot and I also appreciate his valuable suggestions related to career decisions. Prof. Zhiqiang Lin has always been a good teacher and a kind friend. His positive academic attitude has always inspired me. I wish I could become a researcher like Prof. Lin in the future.

My parents, Liuyi Tang and Zhiwu Li, and my girlfriend, Yizhi Wang are the most important part of my entire life and I could not finish my Ph.D. degree without their support and understanding. The company of my girlfriend makes my life more colorful and vibrant, which relaxes me whenever I feel tired. While I learn academic knowledge from professors and peers, my optimistic attitude and careful habits are closely related to my parents' childhood education, which works as the cornerstone of a building and plays an irreplaceable role in past, now and future. I really miss those time spent with my family.

I am also grateful for those close colleagues in my life. Xiaokuan Zhang, Guoxin Chen, Yuan Xiao, Luca Wilke, Shixuan Zhao, Xin Jin, Huibo Wang, Chaoshun Zuo, Wubing Wang, Qingchuan Zhao, Sanchuan Chen, Mengya Zhang, and Haohuang Wen, who study in the same lab room with me, have uncountable inspiring and exciting discussions together and also help me a lot in my daily life. All my roommates in the last five years, Xiang Gu, Yifan Gan, Yujie Hui, Jiaxu Qu, Juncheng Zhou, Shuaichen Zhou, Yi Man, and Bo Qiao are all my best friends. I have many precious memories with you guys and I hope we could all live and grow in our desired styles. My best friend since primary school, Lingke Cheng, always supports and inspires me. Even though we have not seen each other for a while, I still can sense the close connection between us.

To present my sincerely thanks to everyone again.

Vita

2016 ..... B.S. Information Engineering,
Shanghai Jiao Tong University, Shanghai, China.

2016 - present ..... Ph.D. Candidate,
The Ohio State University, Ohio, USA.

Publications

Selected Research Publications

Mengyuan Li*, Luca Wilke*, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, Yinqian Zhang. A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP. In *2022 IEEE Symposium on Security and Privacy (**S&P**)*, 2022.

Shixuan Zhao, **Mengyuan Li**, Yinqian Zhang, Zhiqiang Lin. vSGX: Virtualizing SGX Enclaves on AMD SEV. In *2022 IEEE Symposium on Security and Privacy (**S&P**)*, 2022.

Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, Yueqiang Cheng. CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel. In *30th USENIX Security Symposium (**USENIX Security**)*, 2021.

Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, Yueqiang Cheng. TLB Poisoning Attacks on AMD Secure Encrypted Virtualization. In *Annual Computer Security Applications Conference (**ACSAC**)*, 2021.

Mengyuan Li, Yinqian Zhang, Zhiqiang Lin. CROSSLINE: Breaking "Security-by-Crash" based Memory Isolation in AMD SEV. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (**CCS**)*, 2021.

Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, Yan Solihin. Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization. In *28th USENIX Security Symposium (**USENIX Security**)*, 2019.

Liang Wang, **Mengyuan Li**, Yinqian Zhang, Thomas Ristenpart, Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (**USENIX ATC**)*, 2018.

Guoxing Chen, **Mengyuan Li**, Fengwei Zhang, and Yinqian Zhang, "Defeating Speculative-Execution Attacks on SGX with HyperRace". In *2019 IEEE Conference on Dependable and Secure Computing (DSC)*, 2019.

Yuan Xiao, **Mengyuan Li**, Sanchuan Chen, Yinqian Zhang. Stacco: Differentially analyzing side-channel traces for detecting SSL/TLS vulnerabilities in secure enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (**CCS**)*, 2017.

Mengyuan Li, Yan Meng, Junyi Liu, Haojin Zhu, Xiaohui Liang, Yao Liu, Na Ruan. When CSI meets public WiFi: inferring your mobile phone password via WiFi signals. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security (**CCS**)*, 2016.

Haojin Zhu, Chenliaohui Fang, Yao Liu, Cailian Chen, **Mengyuan Li**, Xuemin Sherman Shen. You can jam but you cannot hide: Defending against jamming attacks for geo-location database driven spectrum sharing. In *IEEE Journal on Selected Areas in Communications 34, no. 10 (2016): 2723-2737*.

Fields of Study

Major Field: Computer Science and Engineering

Table of Contents

Page

Abstract ii

Acknowledgments v

Vita vii

List of Tables xiii

List of Figures xv

List of Listings xvii

1. Introduction 1

    1.1 Overview 1
    1.2 I/O Security in SEV 3
    1.3 ASID Security in SEV 4
    1.4 TLB Security in SEV 7
    1.5 VMSA Security in SEV 9
    1.6 General Memory Encryption Security in SEV 11

2. Background 13

    2.1 AMD SEV 13
        2.1.1 Overview 13
        2.1.2 Memory Encryption 17
    2.2 Existing Security Studies on SEV 19
    2.3 Comparison between Intel TEEs and SEV 23

3. Exploiting Unprotected I/O Operations in AMD SEV 26

3.1 Root Cause of vulnerable I/O operations 27
3.2 Threat Model 28
3.3 Security issues of Unprotected I/O Operations in AMD SEV 29
3.3.1 Unprotected I/O Security's consequences 29
3.3.2 Decryption Oracles 33
3.3.3 Encryption Oracle 41
3.4 Evaluation 43
3.4.1 Pattern Matching 44
3.4.2 Persistent B_p 45
3.4.3 I/O Performance Degradation 46
3.4.4 An End-to-End Attack 48
3.5 A Path Towards I/O Security in SEV 49
3.5.1 Authenticated Encryption 51
3.5.2 A Temporary Software Solution 54
3.5.3 Comparison with existing attacks 56
3.6 Summary 56

4. Breaking "Security-by-Crash" based Memory Isolation in AMD SEV 58

4.1 Demystifying ASID-based Isolation 59
4.1.1 ASID-based Isolation 59
4.1.2 ASID Management 62
4.1.3 ASID Isolation Summary 66
4.2 CROSSLINE Attacks 67
4.2.1 Variant 1: Extracting Encrypted Memory through Page Table Walks 68
4.2.2 Variant 2: Executing Victim VM's Encrypted Instructions 74
4.2.3 Discussion on Stealthiness and Robustness 78
4.3 Applicability to SEV-ES 79
4.3.1 Overview of SEV-ES 79
4.3.2 CROSSLINE V1 on SEV-ES 81
4.3.3 CROSSLINE V2 on SEV-ES 86
4.3.4 Discussion on Stealthiness 87
4.4 Discussion 88
4.4.1 A New Variant: Reusing Victim's TLB Entries 88
4.4.2 Applicability to SEV-SNP 89
4.4.3 Intel MKTME 90
4.4.4 Relation to Speculative Execution Attacks 91
4.5 Summary 92

5. TLB Poisoning Attacks on AMD Secure Encrypted Virtualization 93

5.1 Background 94
5.2 Understanding and Demystifying SEV's TLB Isolation Mechanisms . . . 96
5.2.1 TLB Management for Non-SEV VMs 96
5.2.2 Demystifying SEV's TLB management 98
5.2.3 TLB Flush Rules for SEV VMs 100
5.3 Attack Primitives 101
5.3.1 Threat Model 101
5.3.2 TLB Misuse across vCPUs 102
5.3.3 TLB Misuse within the Same vCPU 104
5.3.4 CPUID-based Covert Channel 106
5.4 TLB Poisoning with Assisting Processes 109
5.4.1 Case Study: OpenSSH 109
5.4.2 Evaluation 112
5.5 TLB Poisoning without Assisting Processes 112
5.5.1 Poison TLB Entries between Connections 113
5.5.2 An End-to-end Attack 115
5.5.3 Evaluation. 118
5.6 Discussion and Countermeasure 120
5.6.1 TLB Poisoning on SEV-SNP 120
5.6.2 Comparison with Known Attacks 121
5.6.3 Countermeasures 122
5.7 Summary 122

6. CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel 123

6.1 Background 124
6.1.1 Secure Encrypted Virtualization 124
6.1.2 Cryptographic Side-Channel Attacks 127
6.1.3 Advanced Programmable Interrupt Controller 129
6.2 The CIPHERLEAKS Attack 130
6.2.1 The Ciphertext Side Channel 130
6.2.2 Execution State Inference 133
6.2.3 Plaintext Recovery 135
6.3 Case Studies 142
6.3.1 Breaking Constant-Time RSA 142
6.3.2 Breaking Constant-time ECDSA 144
6.3.3 Evaluation 146
6.4 Countermeasures 148

6.4.1 Software Mitigation . . . 148
6.4.2 Function's Internal States Intercept . . . 149
6.4.3 Hardware Countermeasures . . . 152
6.5 Applicability to SEV-SNP . . . 153
6.5.1 Overview of SEV-SNP . . . 154
6.5.2 The CIPHERLEAKS attack on SEV-SNP . . . 155
6.6 Summary . . . 156

7. A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP . . . 157

7.1 Background . . . 158
7.1.1 Secure Encrypted Virtualization . . . 158
7.1.2 Ciphertext Attacks against SEV-SNP . . . 160
7.1.3 Off-chip Attacks . . . 161
7.1.4 Operating System Context Switch . . . 161
7.2 A generic ciphertext side channel . . . 162
7.2.1 Attacker Model . . . 162
7.2.2 Attack Primitives . . . 163
7.3 Leakage due to context switch . . . 166
7.3.1 Leaking Register Values via Context Switches . . . 166
7.3.2 Attacking Constant-time ECDSA . . . 169
7.3.3 End-to-end attack against Nginx . . . 171
7.3.4 Evaluation . . . 172
7.4 Exploiting memory accesses in user space . . . 174
7.4.1 Breaking Constant-time ECDSA via Dictionary Attack . . . 175
7.4.2 Breaking Constant-time EdDSA via collision attack . . . 178
7.5 Countermeasures . . . 183
7.5.1 Architectural Countermeasures . . . 183
7.5.2 Software-based Countermeasures . . . 184
7.5.3 Software-based Countermeasures: Kernel Context Switch . . . 186
7.5.4 Case Study: Randomizing pt_regs Location . . . 188
7.6 Discussion . . . 189
7.7 Summary . . . 192

8. Conclusion . . . 193

Bibliography . . . 194

List of Tables

Table **Page**

2.1 Effects of C-bits in guest page tables (gPT) and nested page tables (nPT). M is the plaintext; $E_k()$ is the encryption function under a memory encryption key k; k_g and k_h represent the guest VM and the hypervisor's memory encryption keys, respectively. 17

3.1 End-to-end attack performance. 49

4.1 Demonstrated attacks against SEV. I/O Interaction: the attack requires interaction with applications inside the victim VM through I/O operations (*e.g.*, Network, disk). Stealthiness: the attack cannot be detected by the victim VM. 87

5.1 TLB flush rules. The *World* column indicates whether the event happens in host world or the guest world; *TLB tag* represents the TLB entry's ASID to be flushed—the host's ASID is 0 and the SEV VM's ASID is N; *Forced* indicates whether the TLB flush is forced by the hardware or controllable by the hypervisor. * highlights a special case, in which when the world switch happens between two vCPUs, the TLB tagged with 0 is forced to be flushed while the TLB tagged with N is flushed under the control of the hypervisor. 100

6.1 Ciphertext of registers collected in the VMSA. If the content at a specific offset is 8 bytes, it means the remaining 8 bytes are reserved. 132

6.2 Information revealed from NPF error code. 134

6.3 Number of NAE events observed during boot period and registers state range maybe exposed. Num: the number of NAE event being observed. *: state to hypervisor. **: state from hypervisor, N/A: not observed. -: this register is not supposed to be used during this NAE event. Range R1: numbers of different exposed register states lying in [0,1], Range R2: [0,15], Range R3: [0,127], Range R4: [0,2^{64}-1]. 139

7.1 Possible `pbit` and `kbit` pairs when intercepting `BN_is_bit_set()`. The letters `A` to `D` represent the 16-byte ciphertexts the attacker may observe, which depend on the values of `kbit` and `pbit`. The value of `kbit` and `pbit` in the $i+1$-th iteration is updated depending on k_i. 177

7.2 Comparison of hardware memory encryption-based TEEs. Drop-In replacement means that applications do not need to be adjusted to work with the TEE. * denotes the release time of the whitepapers while the commercial machine is not available yet. † denotes the new SGX version (SGX on Ice Lake SP). 192

List of Figures

Figure **Page**

2.1 AMD-V nested page table walks [3]. 15

3.1 An example of a disk I/O operation by an SEV-enabled VM. 28

3.2 Read/write performance overhead due to I/O monitoring. 31

3.3 A decryption oracle. Step ①, the hypervisor conducts pattern matching using page-fault side channels to determine the address of B_p. Step ②, the hypervisor replaces a ciphertext block in B_p with the target memory block, which will be decrypted when copied to B_s. Step ③, QEMU recovers the network packet headers. 35

3.4 Format of an SSH packet. 39

3.5 An encryption oracle. Step ①, QEMU forwards an incoming packet to the guest. Step ②, QEMU passes the address of B_s to the hypervisor. Step ③, a page fault immediately after the fault at B_s is captured by the page fault handler. Step ④, message m' is placed in B_p. Step ⑤, page fault handler returns the control to the guest. 43

3.6 The precision and recall of determining P_{priv} in 20 rounds or experiments. . 45

3.7 Reduction of incorrect guesses using the N-Streak strategy. 46

3.8 The number of different B_ps used with various rates of packets (pps). . . . 47

3.9 Statistics of repeated B_ps. 48

3.10 I/O performance degradation evaluated using SSH response time (network latency excluded). 49

3.11 Merkle Tree (a) and Bonsai Merkle Tree used in conjunction with split counter mode encryption (b). 52

3.12 Counter mode encryption with split counters using Galois Counter Mode authentication. 53

3.13 An illustration of the temporary software solution. 55

4.1 ASID-based memory isolation in SEV. 60

4.2 Workflow of CROSSLINE V1. 68

4.3 Valid PTE format. 73

4.4 Covered offsets after *N* rounds. 85

5.1 TLB misuses across vCPUs. 102

5.2 VMCB switching. 108

5.3 Attack steps to bypass password authentication. 116

5.4 Variation of the virtual address of `testcrypto`. 119

6.1 Example about the ciphertext changes in NPFs. 136

6.2 Workflow of how VC handler handles IOIO_PROT events. 138

6.3 Performance of stepping VM execution using APIC. 152

6.4 The RMP Check in AMD-SNP. 155

7.1 Encryption block configurations with different exploitability by the dictionary attack. In the first scenario (a), most of the block's plaintext is constant, with the secret being the only variable. Thus, the attacker can build a one-to-one mapping of ciphertexts to secrets. In (b), the block also contains a loop counter *i*, so there are many different ciphertexts mapping to the same secret. If the attacker can always observe the secret for a specific fixed value of *i*, they may still be able to build a dictionary, as this is equivalent to scenario (a). In the last scenario (c), the secret is followed by a random nonce which is regenerated before spilling secret to the memory. This prevents the attacker from creating a dictionary, as he never observes the same ciphertext twice. 164

7.2 Workflow of how #VC exceptions are handled. Red arrows represent a context switch between processes. 168

7.3 Relationship between `udelay` interval and internal context switch. 174

List of Listings

Listing **Page**

5.1 Code snippet of `pre_sev_run()`. 98

5.2 Code snippet of `svr_auth_password()`. 114

6.1 Code snippet of `BN_mod_exp_mont_consttime`. 143

6.2 Code segment of `bn_get_bits5()`. 144

6.3 Code snippet of `ec_scalar_mul_ladder()`. 145

6.4 Assembly code snippet of `BN_is_bit_set()`. 146

7.1 Part of the elliptic curve scalar multiplication `ec_scalar_mul_ladder()` from OpenSSL. The function uses the Montgomery ladder algorithm and constant-time primitives to protect the secret scalar k against side channels. 170

7.2 Function performing the multiplication of the secret scalar with the curve base point. In the original code, the variable k is named s. 180

7.3 Swap and lookup table access functions. 181

Chapter 1: Introduction

1.1 Overview

Secure Encrypted Virtualization (SEV) is an emerging processor feature available in recent AMD processors that encrypts the entire memory of virtual machines (VM) transparently. Memory encryption is performed by a hardware memory encryption engine (MEE) embedded in the memory controller that encrypts memory traffic on the fly, with a key unique to each of the VMs. As the encryption keys are generated from random sources at the time of VM launches and are securely protected inside the secure processor in their lifetime, privileged software, including the hypervisor, is not able to extract the keys and use them to decrypt the VMs' memory content. Therefore, SEV enables a stronger threat model, where the hypervisor is removed from the trusted computing base (TCB).

> *"SEV technology is 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."* [51].

Hence, SEV provides a trusted execution environment (TEE) for (mostly) unmodified VMs to perform confidential computation that is shielded from strong adversaries that control the entire privileged software stack.

The lack of trust in the hypervisor, unfortunately, increases considerably the attack surface that a VM has to guard against. SEV's audacious threat assumption has been

examined under the microscope with numerous attacks (e.g., [23, 29, 39, 61, 68, 69, 91]) since its debut in 2017. With the assumption of a malicious hypervisor, these attacks successfully compromise the confidentiality and/or integrity provided by SEV's memory encryption by exploiting a number of design flaws, including unencrypted virtual machine control blocks (VMCB) [39, 91], unauthenticated memory encryption [23, 29, 39], insecure ECB mode of memory encryption [29] and unprotected nested page tables [68, 69]

In light of these security issues, AMD has enhanced SEV with a sequence of microcode and hardware updates, most notably SEV with Encrypted State (SEV-ES) and SEV with Secure Nested Paging (SEV-SNP). SEV-ES encrypts the VMCB of a VM to protect register values at VMEXITs; SEV-ES processors are already commercially available. To address the most commonly exploited flaw—the lack of memory integrity for SEV VMs (including unauthenticated memory encryption and unprotected nested page tables), AMD plans to release SEV-SNP, which introduces a Reverse Map Table (RMP) to dictate ownership of the memory pages, so that the majority of the previously known attacks will be mitigated.

SEV and its updated version are AMD's ambitious movement towards confidential cloud computing, which is gaining traction in the cloud industry. For instance, Google Cloud recently provides SEV-enabled VMs, called Confidential VMs, as its first product of Confidential Computing [33]. The threats SEV is facing are essential for SEV customers to understand their data safety in the cloud.

In this proposal, we pay our attention to several, yet-to-be-reported several vulnerabilities. In this proposal, we aim to exploit several vulnerabilities in AMD SEV, including I/O security (Chapter 3), asid-based encryption security (Chapter 4), TLB security (Chapter 5), context switch security (Chapter 6), and general memory encryption security (Chapter 7).

1.2 I/O Security in SEV

In Chapter 3, we study a previously unexplored problem under SEV's trust model—the unprotected I/O operations of SEV-enabled VMs. While the entire memory of the VMs can be encrypted using keys that are not known to the hypervisor, direct memory access (DMA) from the virtualized I/O devices must operate on *unencrypted memory* or *memory shared with the hypervisor*. As a result, neither the confidentiality nor the integrity of the I/O operations can be guaranteed under SEV's trust models.

More importantly, in Chapter 3, we go beyond the investigation of I/O insecurity itself. In particular, we further demonstrate that these unprotected I/O operations can be leveraged by the adversary to construct (1) an encryption oracle to encrypt arbitrary memory blocks using the guest VM's memory encryption key, and (2) a decryption oracle to decrypt any memory pages of the guest. We demonstrate in the chapter that these two powerful attack primitives can be constructed in a very stealthy manner—the oracles can be queried by the adversary repeatedly and frequently without crashing the attacked VMs.

In addition, as a by-product of the study, Chapter 3 also reveals a severe side-channel vulnerability of SEV: As the adversary is able to manipulate the nested page tables, it could alter the `present` bit or `reserve` bit of the nested page table entries to force guest VM's memory accesses to the corresponding pages to trigger page faults. While this page-fault side channel has been previously studied in the context of Intel SGX [95] and even used in previous attacks against SEV [39], it is also reported that page faults from SEV-enabled guest VMs leak the entire faulting addresses (and error code) to the hypervisor (unlike in SGX where the page offset is masked). This fine-grained page-fault attack enables fine-grained tracing of the encrypted VM's memory access patterns, and particularly is used to facilitate the construction of the memory decryption oracle.

Contribution. Our work in Chapter 3 contributes to the study of trusted execution environment (TEE) security in the following aspects:

- Our work study a previously unexplored security issue of AMD SEV—the unprotected I/O operations of SEV-enabled guest VMs. The root cause of the problem is the incompatibility between AMD-V's I/O virtualization with SEV's memory encryption scheme.
- Our work demonstrate that the unprotected I/O operations could also be exploited to construct powerful attack primitives, enabling the adversary to perform arbitrary memory encryption and decryption.
- Our work report the lack of page-offset masking of the faulting addresses during SEV's page faults handling, which leads to fine-grained side-channel leakage. This chapter also demonstrates the use of both fine-grained and coarse-grained side channels in its I/O attacks.
- Our work empirically evaluate the fidelity of the attacks and discusses both hardware and software approaches to mitigating the I/O security issues.

1.3 ASID Security in SEV

In Chapter 3, we show that the unprotected I/O operation may break SEV's integrity and confidentiality. In Chapter 4, we move our attention to another, yet-to-be-reported design flaw of SEV—the improper ASID-based memory isolation and access control. Specifically, SEV adopts an ASID-based access control for guest VMs' accesses to SEV processor's internal caches and the encrypted physical memory. At launch time, each SEV VM is assigned a unique ASID, which is used as the tag of cache lines and translation lookaside buffer (TLB) entries. A secure processor (dubbed AMD-SP) that is in charge of generating and maintaining the ephemeral memory encryption keys also uses the current VM's ASID

to index the keys for encrypting/decrypting memory pages upon memory access requests. As such, the ASID of an SEV VM plays a critical role in controlling its accesses to the private data in the cache-memory hierarchy. Nevertheless, the assignment of ASID to a VM is under complete control of the hypervisor. An implicit "security-by-crash" security principle is adopted in the SEV design:

> *"Although the hypervisor has control over the ASID used to run a VM and select the encryption key, this is not considered a security concern since a loaded encryption key is meaningless unless the guest was already encrypted with that key. If the incorrect key is ever loaded or the wrong ASID is used for a guest, the first instruction fetch of that guest will fail as memory will be decrypted with the wrong key, causing junk data to be executed (and very likely causing a fault)."* [51]

The aim of this chapter is to investigate the validity of this "security-by-crash" design principle. To do so, we first study how ASIDs are used in SEV processors to isolate encrypted memory pages as well as CPU caches and TLBs. We also explore how ASIDs are managed by the hypervisor, how an ASID of a VM can be altered by the hypervisor at runtime, and why the VM with altered ASID crashes afterwards. This exploration leads to the discovery of several potential opportunities for a VM with an altered ASID to *momentarily* breach the ASID-based memory isolation before it crashes.

Next, based on our exploration, we then present CROSSLINE attacks[2], which exploit such a *momentary execution* to breach the confidentiality and integrity of SEV VMs. Specifically, an adversary controlling the hypervisor can launch an attacker VM and, during its VMEXIT, assign it with the same ASID as the victim VM, and then resume it, leading to the violation of the ASID-based access control to the victim's encrypted memory.

We mainly present two variants of CROSSLINE. In CROSSLINE V1, even though no instructions are executed by the attacker VM after VMRUN, we show that it is possible to

[2]CROSSLINE refers to interference between telecommunication signals in adjacent circuits that causes signals to cross over each other.

load memory pages encrypted with the victim VM's memory encryption key (VEK) during page table walks, thus revealing the encrypted content of the "page table entries" (PTE) through nested page faults. This attack variant enables the adversary to extract the entire encrypted page table of the SEV guest VM, as well as any memory blocks conforming to the PTE format. We have also successfully demonstrated CROSSLINE V1 on SEV-ES machines, in which we devise techniques to bypass the integrity checks of launching the attacker VM with the victim VM's encrypted VMCB, while keeping the victim VM completely unaffected. In CROSSLINE V2, by carefully crafting its nested page tables, the attacker VM could manage to momentarily execute arbitrary instructions of the victim VM. By wisely selecting the target instructions, the adversary is able to construct encryption oracles and decryption oracles, which enable herself to breach both integrity and confidentiality of the victim VM. CROSSLINE V2 is confined by SEV-ES, but its capability is stronger than V1.

As extensions of the two attack variants, in Chapter 4, we also discuss (1) another variant of CROSSLINE, which allows the attacker VM to reuse the TLB entries of the victim VM for address translation and execute some instructions, even without any successful page table walks; and (2) the potential applicability of CROSSLINE on SEV-SNP.

Differences from known attacks. CROSSLINE differs from all previously demonstrated SEV attacks in several aspects. *First*, CROSSLINE does not rely on SEV's memory integrity flaws, which is a common pre-requisite for all known attacks on SEV. Although CROSSLINE may not work on SEV-SNP, the protection does not come from memory integrity, but a side-effect of the RMP implementation. *Second*, CROSSLINE attacks do not directly interact with the victim VMs and thus enable *stealthy* attacks. As long as the ephemeral encryption key of the victim VM is kept in the AMD-SP and the victim's encrypted memory pages are not deallocated, CROSSLINE attacks can be performed even when the victim VM is

shutdown. Therefore, CROSSLINE is undetectable by the victim VM. In contrast, prior attacks relying on I/O operations of the victim VM [29, 61, 68, 69] are detectable by the victim VM.

CROSSLINE questions a fundamental security assumption of "security-by-crash" underpinning the design of SEV's memory and cache isolation. The demonstration of these attack variants suggests that SEV should not rely on adversary-controlled ASIDs to mediate access to the encrypted memory. To eliminate the threats, a principled solution is to maintain the identity of VMs in the hardware, which unfortunately requires some fundamental changes in the architecture. As far as we know, SEV-SNP will not integrate such changes.

Contributions. This chapter makes the following contributions to the security of AMD SEV and other trusted execution environments.

- Our work investigate SEV's ASID-based memory, cache, and TLB isolation, and demystifies its "security-by-crash" design principle (Section 4.1). It raises security concerns of the "security-by-crash" based memory and TLB isolation for the first time.
- Our work present two variants of CROSSLINE attacks—the only attacks that breach the confidentiality and integrity of an SEV VM without exploiting SEV's memory integrity flaws (Section 4.2).
- Our work presents successful attacks against SEV and SEV-ES processors (Section 4.3). We also discusses the applicability of CROSSLINE on the upcoming SEV-SNP processors (Section 5.6).

1.4 TLB Security in SEV

Chapter 5 outlines a new category of security attacks against SEV, namely TLB Poisoning Attacks, which enable the adversary who controls the hypervisor to poison the TLB entries

shared between two processes of the same SEV VM. The root cause of TLB Poisoning Attacks is that the hypervisor is in control of the TLB flushes by the design of AMD SEV. Specifically, because TLB is tagged with ASIDs to distinguish the TLB entries used by different entities, unnecessary TLB flushes can be avoided during the world switches (VMEXIT and VMRUN between the guest VM and the hypervisor) or the context switches (context switches between the process hosting the guest VM's current virtual CPU (vCPU) and other processes). As it is difficult for the CPU hardware to determine whether to flush the entire TLB or only TLB entries with certain ASIDs, the TLB flush is solely controlled by the hypervisor. The hypervisor can inform the CPU hardware to fully or partially flush the TLB by setting the TLB control field in the VMCB, which will take effect after VMRUN. As such, the adversary can intentional skip TLB flushes, such that a victim process of the victim SEV VM may unwillingly use the TLB entries injected by another process of the same VM.

Two attack scenarios of TLB Poisoning attacks are considered in this chapter: (1) With the help of an unprivileged attacker process running in the targeted SEV VM, the adversary is able to poison the TLB entries used by a privileged process and alter its execution. (2) Without the help of a process directly controlled by the adversary, the adversary could still exploit the misuse of TLB entries on a network-facing process (not in his control) that share the same (or similar) virtual address space with the targeted process and bypass authentication checks. We have demonstrated two end-to-end attacks against two SSH servers to show the feasibility of the two attack scenarios, respectively, on an AMD EPYC Zen processor that supports SEV-ES.

Contributions. The chapter makes the following contributions to field of study.

- It demystifies AMD SEV's TLB management mechanisms, which have never been studied and reported in-depth, and identifies a severe flaw of its design of TLB isolation that leads to misuse of TLBs under the assumption of a malicious or compromised hypervisor.
- It presents a novel category of attacks against SEV, namely TLB Poisoning Attacks, which manipulate the TLB entries shared by two processes within the same SEV VM and breach the integrity and confidentiality of one of the processes. To the best of our knowledge, it is the first TLB poisoning attack demonstrated in any context.
- It demonstrates two end-to-end TLB Poisoning Attacks against SEV-ES-protected VMs. In one attack, it shows the feasibility of poisoning TLB entries to change the code execution of the victim process; in the other, it provides an example of stealing secret data from the victim process by a process (not controlled by the adversary) through shared TLB entries.

1.5 VMSA Security in SEV

Unlike all prior work on SEV attacks, Chapter 6 presents a new side channel on SEV (including SEV-ES and SEV-SNP) processors. We call it the ciphertext side channel. It allows the privileged hypervisor to monitor the changes of the ciphertext blocks on the guest VM's memory pages and exfiltrate secrets from the guest. The root cause of the ciphertext side channel are two-fold: First, SEV's memory encryption engine uses an XOR-Encrypt-XOR (XEX) mode of operation, which encrypts each 16-byte memory block independently and preserves the one-to-one mapping between the plaintext and ciphertext pairs for each physical address. Second, the design of SEV does not prevent the hypervisor from reading the ciphertext of the encrypted guest memory, thus allowing its monitoring of the ciphertext changes during the execution of the guest VM.

To demonstrate the severity of leakage due to the ciphertext side channel, we construct the CROSSLINE attack, which exploits the ciphertext side channel on the encrypted VMSA page of the guest VM. Specifically, the CROSSLINE attack monitors the ciphertext of the VMSA area during VMEXITs, then (1) by comparing the ciphertext blocks with the ones observed during previous VMEXITs, the adversary is able to learn that the corresponding register values have changed and thereby infer the execution state of the guest VM; and (2) by looking up a dictionary of plaintext-ciphertext pairs collected during the VM bootup period, the adversary is able to recover some selected values of the registers. With these two attack primitives, we show that the malicious hypervisor may leverage the ciphertext side channel to steal the private keys from the constant-time implementation of the RSA and ECDSA algorithms in the latest OpenSSL library, which are believed to be immune to side channels.

We discuss countermeasures of the ciphertext side channel and the specific CROSSLINE attack. While there are some seemingly feasible software countermeasures, we show they become fragile when the CROSSLINE attack is performed using Advanced Programmable Interrupt Controller (APIC). Therefore, we conjecture that the ciphertext side-channel vulnerability is difficult to eradicate from the software. Therefore, alternative hardware solutions must be adopted in the future SEV hardware.

Contributions. This chapter contributes to the security of AMD SEV and confidential computing technology in general in the following aspects:

- It presents a novel ciphertext side channel on SEV processors. This discovery identifies a fundamental flaw in the SEV's use of XEX mode memory encryption.

- It presents a new CipherLeaks attack that exploits the ciphertext side channel to infer register values from encrypted VMSA. Two primitives were constructed for inferring the execution states of the guest VM and recovering specific values of the registers.
- It presents successful attacks against the constant-time RSA and ECDSA implementation of the latest OpenSSL library, which has been considered secure against side channels.
- It discusses the applicability of the CipherLeaks attack on SEV-SNP. To the best of our knowledge, the CipherLeaks attack is the only working attack against SEV-SNP that breaches the memory encryption of the guest VM.
- It discusses potential software and hardware countermeasures for the ciphertext side channel and the demonstrated CipherLeaks attack.

1.6 General Memory Encryption Security in SEV

The first and only software-based attack that still applied to SEV-SNP is CIPHERLEAKS [62] (as introduced in Chapter 6), a novel side-channel attack where a malicious hypervisor can steal the secret keys of RSA and ECDSA algorithms in the OpenSSL implementation by monitoring the guest VM Save Area (VMSA). Due to its severity, AMD recently released a microcode patch (MilanPI-SP3_1.0.0.5) [11] to mitigate the CIPHERLEAKS attacks. The microcode patch enables the 3rd generation AMD EPYC processors (Milan series) to include a nonce into the encryption of the VMSA area, such that the link between the plaintext and the ciphertext is broken. As such, CIPHERLEAKS attacks against register values in the VMSA are no longer feasible. Note that the patch only changes the encryption of the VMSA, while the remaining memory space of the VM is still protected with the same deterministic XEX encryption as before.

In Chapter 7, we perform a comprehensive study on the exploitability of leakage caused by ciphertext in encrypted VM memory and try to answer the question:

> *Are current cryptography implementations still safe when an attacker has access to the ciphertext of the encrypted memory?*

We broadly group ciphertext side channel attacks into two categories: *the dictionary attack* and *the collision attack*. We show that these two classes of attacks can be applied to general memory regions during cryptographic activities, including kernel data structures, stacks, and heaps, which all lead to key leakage. Most main cryptography libraries (including OpenSSL, WolfSSL, GnuTLS, OpenSSH, and libgcrypt) are shown to be vulnerable against the ciphertext side channel.

Contribution. The contributions of this chapter can be summarized as follows:

- Systematically studies the ciphertext side channel in the entire memory of SEV-protected VMs. It shows that the ciphertext side channel can be exploited in *all* memory regions, including kernel structures, stacks, and heaps.
- Presents end-to-end ciphertext side-channel attacks against the ECDSA implementation of the OpenSSL library. Other main cryptography libraries (including OpenSSL, WolfSSL, GnuTLS, OpenSSH, and libgcrypt) are also shown to be vulnerable to the ciphertext side channel.
- Discusses both hardware and software countermeasures. Presents a kernel patch to mitigate ciphertext side channels caused by kernel structures. The ciphertext side channel can be mitigated when adopting the kernel patch together with software fixes for cryptographic libraries.

Chapter 2: Background

2.1 AMD SEV

2.1.1 Overview

AMD Secure Encrypted Virtualization (SEV) is a security extension for AMD Virtualization (AMD-V) architecture [6]. AMD-V is designed as a virtualization substrate for cloud computing services, which allows one physical server to run multiple isolated guest virtual machines (VM) concurrently. AMD's SEV is designed atop its Secure Memory Encryption (SME) technology.

Secure Memory Encryption (SME). SME is AMD's x86 extension for real-time main memory encryption, which is supported in AMD CPU with Zen micro architecture from 2017 [83]. Aiming to defeat cold boot attack and DRAM interface snooping, an embedded Advanced Encryption Standard (AES) engine encrypts data when the processor writes to the DRAM and decrypts it when processor reads it. The entire DRAM is encrypted with a single ephemeral key which is randomly generated each time the machine is booted. A 32-bit ARM Cortex-A5 Secure Processor (AMD-SP) [74] is integrated in the system-on-chip (SOC) alongside the main processor, providing a dedicated security subsystem, storing, and managing the ephemeral key. Although all memory pages are encrypted by default, the operating system can mark some pages as unencrypted by clearing the *C-bit* (the 48th bit) of

the corresponding page table entries (PTE). However, regardless of the C-bit, all code pages and page table pages are encrypted by default. With Transparent SME (TSME), a special mode of operation of SME, the entire memory is encrypted, ignoring the C-bits of the PTEs.

AMD Virtualization (AMD-V). AMD-V is a set of extensions of AMD processors to support virtualization. Nested Page Tables (nPT) is introduced by AMD-V to facilitate address translation, which is officially marketed as Rapid Virtualization Indexing [3]. AMD-V's nPT provides two levels of address translation. When nPT is enabled, the guest VM and the hypervisor have their own CR3s: a guest CR3 (gCR3) and a nested CR3 (nCR3). The gCR3 contains the guest physical address of the gPT; the nCR3 contains the system physical address of the nPT. To translate a virtual address (gVA) used by the guest VM into the system physical address (sPA), the processor first references the guest page table (gPT) to obtain the guest physical address (gPA) of each page-table page. To translate the gPA of each page, an nPT walk is performed. During the nPT walk, the gPA is treated as host virtual address (hVA) and translated into the sPA using the nPT. These address translation steps are illustrated in Figure 2.1.

Translation lookaside buffers (TLB) and Page Walk Cache (PWC) are internal buffers in AMD processors for speeding up the address translation. AMD-V also relies on these internal buffers for performance improvements. AMD-V further introduces an nTLB for nPT. A successful nPT walk caches the translation from gPA to sPA in the nTLB for fast accesses [6], while the normal TLBs are used to store translations from virtual addresses of either the host or the guest to sPA.

To exchange data between the hypervisor and the guest VMs, a data structure dubbed the virtual machine control block (VMCB) is located on a shared memory page. VMCB

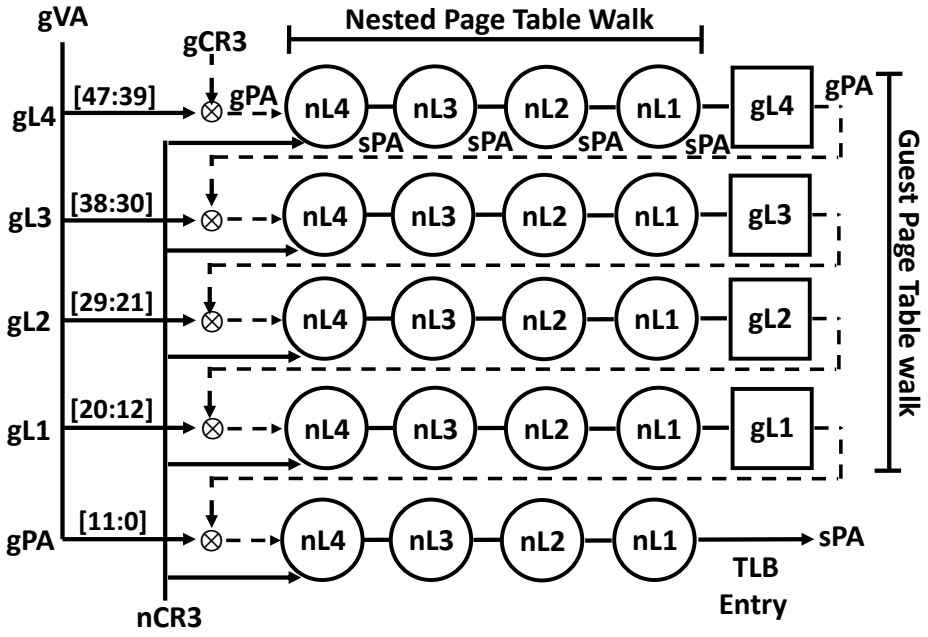


Figure 2.1: AMD-V nested page table walks [3].

stores the guest VM's register values and some control bits during VMEXIT. The VMCB is under the control of the hypervisor to configure the behaviors of the guest VM.

Secure Encrypted Virtualization (SEV). SEV combines AMD-V architecture with SME to allow individual VMs to have their own VM Encryption Key (VEK) [7]. Each VEK is generated by the processor and assigned to an SEV VM when launched by the hypervisor. All VEKs are stored in the AMD-SP and are never exposed to DRAM during their entire life cycle. SEV distinguishes different VEKs using ASIDs. Specifically, when a VM is launched, the hypervisor chooses an available ASID and assigns the ASID to the VM. The ASID is then used as the key index to identify the specify the VEKs inside the co-processor. There is a special register storing current ASID of the VM and the ASID of the hypervisor is restricted by the processor to be 0 [6].

When a memory request is made, the AMD-SP determines which key to be used with the current ASID. In combination with encryption modes specified in the guest page tables (gPT) and the nested page tables (nPT) [3], SEV achieves page-granular memory encryption with different keys. The confidentiality of SEV-enabled VM is guaranteed because memory encryption keys are managed by the AMD-SP, privileged software layers like hypervisor are not allowed to access or manipulate these keys.

Beside confidentiality, authenticity of the platform and integrity of the guest VMs are also provided by SEV. An identification key embedded in the firmware is signed by both AMD and the owner of the machine to demonstrate that the platform is an authentic AMD platform with SEV capabilities, which is administered by the machine owner. The initial contents of memory, along with a set of metadata of the VM, can be signed by the firmware so that the users of the guest VMs may verify the identity and the initial states of the launched VMs through remote attestation.

SEV Encrypted State (SEV-ES) and SEV Secure Nested Paging (SEV-SNP). SEV VMs may additionally use the SEV with Encrypted State (SEV-ES) feature or SEV with Secure Nested Paging (SEV-SNP) feature if the processor has corresponding microcode and hardware support [8]. SEV-ES encrypts the VMCB of a VM to protect register values at VMEXITs; SEV-ES processors are already commercially available. SEV-SNP is AMD's plan to address the most commonly exploited flaw—the lack of memory integrity for SEV VMs. SEV-SNP is used to protect encrypted VM from SEV and SEV-ES's existing vulnerabilities like unauthenticated memory encryption and unprotected nested page tables. AMD plans to release SEV-SNP, which introduces a Reverse Map Table (RMP) to dictate ownership of the memory pages, so that the majority of the previously known attacks will

Table 2.1: Effects of C-bits in guest page tables (gPT) and nested page tables (nPT). M is the plaintext; $E_k()$ is the encryption function under a memory encryption key k; k_g and k_h represent the guest VM and the hypervisor's memory encryption keys, respectively.

gPT	**nPT**	
	C-bit=0	**C-bit=1**
C-bit=0	M	$E_{k_h}(M)$
C-bit=1	$E_{k_g}(M)$	$E_{k_g}(M)$

be mitigated. SEV-SNP processors as well as its software updates are not commercially available yet.

2.1.2 Memory Encryption

ASID and memory encryption. In SEV, the encryption keys used for memory encryption are generated from random sources when the VMs are launched. They are securely stored inside the secure processor for their entire life-cycle. Each VM has its own unique memory encryption key K_{vek}, which is indexed by the ASID of the VM. When the VM accesses a memory page that is mapped to its address space with its C-bit set, the memory will be first decrypted using the VM's K_{vek} before loaded into the CPU caches. Data in the caches are stored in plaintext; each cache line, in addition to the regular cache tags, is also tagged by the ASID of the VM. As such, the same physical memory may have multiple copies cached in the hardware caches. AMD does not maintain the consistency of the cache copies with different ASID tags [51].

Encryption with nested paging. AMD-V utilizes nested paging structures [3] to facilitate memory isolation between guest VMs. When the virtual address used by the guest VM (gVA) is to be translated into physical address, it is first translated into a guest physical

addressing (gPA) using the guest page table (gPT), and the gPA is then translated into the host physical address (hPA) using the nested page table (nPT). While gPT is located in guest VM's address space, nPT is controlled directly by the host.

With AMD's SME technology, bit 47 of a PTE is called the *C-bit*, which is used to indicate whether or not the corresponding page is encrypted. When the C-bit of a page is set (*i.e.*, 1), the page is encrypted. As both the gPT and the nPT has C-bits, the encryption state of a page is controlled by the combination of the two C-bits in its PTEs in the gPT and nPT. The effect of C-bits in the gPT and nPT is shown in Table 2.1. To summarize, whenever the C-bit of gPT is set to 1, the memory page is encrypted with the guest VM's encryption key k_g; when the C-bit of gPT is cleared, the C-bit of nPT determines the encryption state of the page: the page is encrypted under the hypervisor's key k_g when C-bit is 1; otherwise the page is not encrypted.

To share memory pages between a guest VM and the hypervisor while preventing physical attacks, it is required to have the memory page's C-bit set to 0 in its gPT and the C-bit set to 1 in its nPT, so that the page is encrypted under the hypervisor's encryption key. The control of C-bit in gPT is managed by guest OS.

SEV Encryption mode. SEV uses AES as its encryption algorithm. The memory encryption engine encrypts data with a 128-bit key using the Electronic Codebook (ECB) mode of operation [29]. Therefore, each 16-byte aligned memory block is encrypted independently. A physical address-based tweak function $T()$ is utilized to make the ciphertext dependent of not only the plaintext but also its physical address [51]. Specifically, the tweak function is defined as $T(x) = \oplus_{x_i=1} t_i$, where x_i is the ith bit of host physical address x, $\oplus$ is the bitwise exclusive-or (*i.e.*, XOR) and t_i ($1 \leq i \leq 128$) is a 128-bit constant vector. The tweak function takes a physical address as an input and outputs a 128-bit value $T(x)$. Therefore,

the ciphertext c of a plaintext m at address m is $c = E_K(m \oplus T(m))$. The tweak function prevents attacker from inferring plaintext by comparing the ciphertext of two 16-byte memory blocks. However, as the constant vectors t_is remain the same for all VMs (and the hypervisor) on the machine, they can be easily reverse engineered by an adversary. One root problem exploited in prior studies on SEV's insecurity is the lack of integrity of its encrypted memory [23, 29, 39, 69].

2.2 Existing Security Studies on SEV

The security issues of AMD's SEV have been placed under the spotlight since its debut. Demonstrated security attacks mainly targets SEV's *unencrypted VMCB* [39] and SME's *unauthenticated memory encryption* [23, 29, 39, 69]. The former issue has been fixed using SEV-ES [49] and the latter could be addressed with integrity protection of the encrypted memory. An implementation bug in the firmware of AMD secure processors have also been reported [28]. But since the issue was not related to a design failure, we leave it as out of scope of the proposal. We detail these related work as follows:

Unencrypted VMCB. Hetzelt and Buhren analyzed the security of SEV from the perspective of unencrypted virtual machine control block (VMCB) [39]. VMCB is a data structure in memory shared by the hypervisor and the guest VM, which stores the values of guest's general purpose registers and control bits for handling virtual interrupts. At the time of `VMEXIT`, a malicious hypervisor may learn the machine state of the guest VM by reading register values stored in the VMCB and subsequently alter their values before `VMRUN` to control the registers of the guest VM. Hetzelt and Buhren [39] exploit unencrypted VMCB using code gadgets in the guest memory (similar to return-oriented programming (ROP) [79]) to arbitrarily read and write encrypted memory in the guest VM. The security

issue caused by unencrypted VMCB, however, has been mitigated by SEV-ES [49], which adds another indirection layer during `VMEXIT` that allows the guest VM to be notified before Non-Automatic Exits (NAE)—exits requiring hypervisor emulation—and prepares a new data structure called Guest Hypervisor Communication Block—a subset of VMCB—to communicate with the hypervisor. The machine states stored in the VMCB are instead encrypted with authentication, such that they are inaccessible from the hypervisor.

Unauthenticated memory encryption. Because SME does not use authenticated encryption schemes, the integrity of the encrypted memory is not protected. As such, malicious hypervisors may alter the ciphertext of the encrypted memory without triggering errors in the decryption process of the guest VMs. Prior studies have demonstrated a variety of approaches to exploit such unauthenticated memory encryption:

- *Chosen plaintext attacks.* Du *et al.* discovered that SME uses Electronic Codebook (ECB) mode of operation in its memory encryption [29], which implies that the same plaintext always leads to the same ciphertext after encryption. As the only security measure is a physical address based tweak function *XOR*ed with the plaintext before encryption, knowledge of the tweak function will enable the adversary to deduce the relationship between the plaintext of two memory blocks (*i.e.*, of 16 bytes) if their ciphertext are the same. Du *et al.* exploit this weakness by constructing a chosen plaintext attack (via an HTTP server installed on the guest VM) and then replace the ciphertext of an `sshd` program with the ciphertext of instructions specified by the adversary (after applying the tweak functions).

  Wilke *et al.* [93] studied the Xor-Encrypt-Xor (XEX) mode of memory encryption of AMD's Epyc 3xx1 series processors, where the tweak function XOR with the plaintext twice, both before and after the encryption. However, in the Epyc 3xx1 processor that was

studied by the authors, the entropy of the tweak functions is only 32 bits, making brute-force attacks practical. It is demonstrated that the adversary who breaks the tweak function can insert some arbitrary 2-byte instruction into encrypted memory with the help of 8MB plaintext-ciphertext pairs. The vulnerability is also caused by the lack of authentication in the memory encryption. Fortunately, the XEX tweak function vulnerability exploited in the paper was fixed in Zen 2 architecture that was released in May, 2019. Therefore, later AMD processors are not affected by this attack.

- *Fault injection attacks.* Buhren *et al.* studied fault injection attacks on a simulated SME implementation [23]. Their work considers a different threat model, which assumes that the adversary is able to conduct physical DMA attacks [18] and also run an unprivileged process on the target OS. The unprivileged process performs Prime+Probe side-channel attacks to trace the execution of the SME protected application and, at the proper moment of a cryptographic operation, utilizes DMA attacks to inject memory faults to infer secret keys (or key components). We believe Buhren *et al.*'s attack against SME can be migrated to SEV as well, which is even easier to conduct as the hypervisor can be assumed to be malicious.

- *Page table manipulation.* Remapping guest pages in the nested paging structures to replay previously captured memory pages was first studied by Hetzelt and Buhren [39]. A similar idea was later demonstrated by Morbitzer *et al.* in SEVered, an attack that by manipulating the nested page table alters the virtual memory of the guest VMs to breach the confidentiality of the memory encryption [69]. More specifically, SEVered is carried out in the following steps: First, the malicious hypervisor sends network requests to the guest's network-facing application, *e.g.*, an HTTP server, which allows the attacker to download files larger than one memory page. Second, using a coarse-grained page-level

side channel, the attacker determines which of the encrypted guest VM's memory pages are used to store the response data. Third, after locating these pages, the malicious hypervisor changes the page mappings in the nested page table so that these virtual pages used by the guest are mapped to different physical pages. As a result, memory content of these pages can be leaked through the responses of the network applications. The same authors further extend SEVered to perform more realistic attacks [68], by extracting secret keys in real-world protocols and applications such as TLS, SSH, full disk encryption (FDE). Their attack makes use of the same side channels to identify the set of memory pages that are likely to contain those secrets and scans those pages (roughly 100 pages) until the secrets are found. Both these works only present decryption oracles but not encryption oracles.

Other studies. Mofrad *et al.* [67] compare Intel SGX and AMD SEV, in terms of their functionality, use scenarios, security, and performance implications. The study suggests SEV is more vulnerable than SGX as it lacks memory integrity and has a bloated trust computing base (TCB). Moreover, the performance comparison suggests AMD SEV technology performs better than Intel SGX. Wu *et al.* proposes Fidelius [94], a system that leverages a sibing-based protection mechanism to partition an untrusted hypervisor into two components, one for resource management and the other for security protection. The security of guest VMs is enhanced by the "trusted" security protection component, which, while interesting and effective, unfortunately contradicts with SEV's original intention of eliminating the hypervisor from the TCB. Fidelius mentioned a method to protect disk I/O that is similar to our temporary fix (see Section 3.5.2) but implies that the disk image is shipped to the SEV platform. Thus it requires using the same K_{tek} every time the disk image is used. Our proxy-style solutions in Section 3.5.2 is a generalization of their approach.

2.3 Comparison between Intel TEEs and SEV

Intel TME and MKTME. Intel's counterparts of AMD's SME and SEV are Total Memory Encryption (TME) and Multi-Key Total Memory Encryption (MKTME) [42]. The concept of TME is almost the same as AMD SME: an AES-XTS encryption engine sits between a direct data path and external memory buses to encrypt data when leaving the processor and decrypt it when entering the processor. TME supports a single ephemeral encryption key for the entire processor. In contrast, MKTME supports multiple keys; it labels each page table entry with a KeyID to select one of the ephemeral AES keys generated in the encryption engine. Different from AMD SEV, guest VMs in MKTME may have more than one AES key. KeyID0 is used for guest VM to share pages with hyperviosr. KeyID*N* is assigned to guest the *N*th VM by hypervisor for guest's private page. However, the guest VM is able to obtain other KeyIDs to share memory with another guest VMs. As we were not able to purchase a machine with TME and MKTME on the market at the time of writing, we leave the analysis of these Intel's technologies to future work.

Intel SGX. Intel Software Guard eXtension (SGX) is an instruction set architecture extension that supports isolation of memory regions of userspace processes. Through a microcode-extended memory management unit, memory accesses to the protected memory regions, dubbed *enclaves*, are mediated so that only instructions belonging to the same enclave are permitted. Software attacks from all privileged software layers, including operating systems, hypervisors, system management software, are prevented by SGX. A hardware Memory Encryption Engine sits between the processor and the memory to encryption memory traffic on the fly, so that confidentiality of the enclave memory is guaranteed even with physical attackers. Remote attestation is supported in SGX to guard the *integrity* of the enclave code.

Similar to AMD's SEV, SGX constructs TEE on Intel processors. However, it differs from SEV as it only isolates portions of the user processes' memory space, whereas SEV encrypts the memory of the entire virtual machine. Developers of SEV do not need to rewrite the software when using AMD's TEE; but SGX developers have to manually partition applications into trusted and untrusted components, and recompile the source code with the SDKs provided by Intel. SGX machines have been available on market since late 2015. So far, two major types of attacks have been demonstrated to SGX applications.

- *Side-channel attacks.* Prior studies have demonstrated that enclave secrets in SGX can be exfiltrated through side channels on the CPU caches [21, 34, 38, 77], branch target buffers [59], DRAM's row buffer contention [90], page-table entries [88, 90], and page-fault exception handlers [80,95]. More recently, side-channel attacks exploiting speculative and out-of-order execution have been shown on SGX as well [26,87]. Similar to SGX, SEV is not designed to thwart side-channel attacks. Therefore, we expect similar attacks can be carried out on AMD's SEV as well. Because some attacks demonstrated in this proposal already completely break the confidentiality of SEV-protected VMs, there is no need to rely on side channels to extract secrets. However, in some of the attacks we demonstrate, side channels do facilitate the attacks. We leave the discussion on side-channel surface of SEV to future work.
- *Memory hijacking attacks.* SGX does not guard memory safety inside the enclaves. Studies [20,58] have shown that attackers could exploit vulnerabilities in enclave programs and perform return-oriented programming (ROP) attacks [79]. Randomization-based security defenses have been proposed to mitigate ROP attacks [78]. However, as pointed out by Biondo *et al.* [20], SGX runtimes inherently contains memory regions that are hard to randomize, and thus completely eliminating the threats of memory hijacking attacks

requires eradicating vulnerabilities from the enclave code. As neither SGX nor SEV is designed to provide memory safety, memory hijacking attacks are feasible on SEV as well. We will not further discuss these attacks on SEV in this proposal.

Chapter 3: Exploiting Unprotected I/O Operations in AMD SEV

In this chapter, we study the insecurity of SEV from the perspective of the unprotected I/O operations in the SEV-enabled VMs. The results are alerting: not only have we discovered attacks that breach the confidentiality and integrity of these I/O operations—which we find very difficult to mitigate by existing approaches—but more significantly we demonstrate the construction of two attack primitives against SEV's memory encryption schemes, namely a memory decryption oracle and a memory encryption oracle, which enables an adversary to decrypt and encrypt arbitrary messages using the memory encryption keys of the VMs. We evaluate the proposed attacks and discuss potential solutions to the underlying problems.

Responsible disclosure. We have reported our findings in this chapter to AMD and disclosed the technical details with AMD researchers. While we were confirmed that the presented attacks work on current release of SEV processors, AMD researchers suggested future generations of SEV chipsets are likely to be immune from these attacks. Some of the technical feedback we obtained from AMD has also been integrated into the chapter.

The chapter is organized as follow: Section 3.1 explains the root causes of the exploited I/O operations. Section 3.2 talks about our threat model. Section 3.3 describes several attacks exploiting the unprotected I/Os. Section 3.4 presents an evaluation of the fidelity of the attacks. Section 3.5 discusses potential solutions to securing SEV's I/O operations and Section 3.6 summarizes the chapter.

3.1 Root Cause of vulnerable I/O operations

The lack of trust in the privileged software introduces an assortment of new attack vectors to SEV-enabled VMs that were mostly unexplored in the literature. In this chapter, we study the unprotected I/O operations in SEV VM. The simulated I/O operations used in SEV and the transmission needed for I/O between shared page and private page are two root causes of vulnerable I/O operations in SEV.

Simulated I/O operations Similar to other virtualization technologies, SEV-enabled VMs interact with I/O devices through virtual hardware using Quick Emulator (QEMU). Common methods for VMs to perform I/O operations are programmed I/O, memory-mapped I/O, and direct memory access (DMA). Among these methods, DMA is most frequently used method for SEV-enabled VMs to do I/O accesses.

Bounce Buffer used in SEV. With the assistance of DMA chips, programmable peripheral devices can transfer data to and from the main memory without involving the processor. With virtualization, a common way to support DMA is through IOMMU, which is a hardware memory management unit that maps the DMA-capable I/O buses to the main memory. However, unique to SEV is that the memory is encrypted. While the MMU supports memory encryption with multiple ASIDs, IOMMU only supports one ASID (*i.e.*, ASID=0). Therefore, in SEV-enabled VMs, DMA operations are performed on memory pages that are shared between the guest and the hypervisor (encrypted with the hypervisor's K_h). A bounce buffer, called Software I/O Translation Buffer (`SWIOTLB`), is allocated on these memory pages.

To illustrate the DMA operation from the guest, a disk I/O read is shown in Figure 3.1. When a guest application needs to read data from file, it first checks whether the file is

already stored in its page cache. A miss in the guest page cache will trigger read from virtual disks, which is emulated by QEMU-KVM. The data is actually read from the physical disk by QEMU-KVM's DMA operation into `SWIOTLB` and then copied to the disk device driver's I/O buffer by the guest VM itself. The disk write operation is the inverse of this process, in which the data is first copied from the guest into `SWIOTLB` and then processed by QEMU.

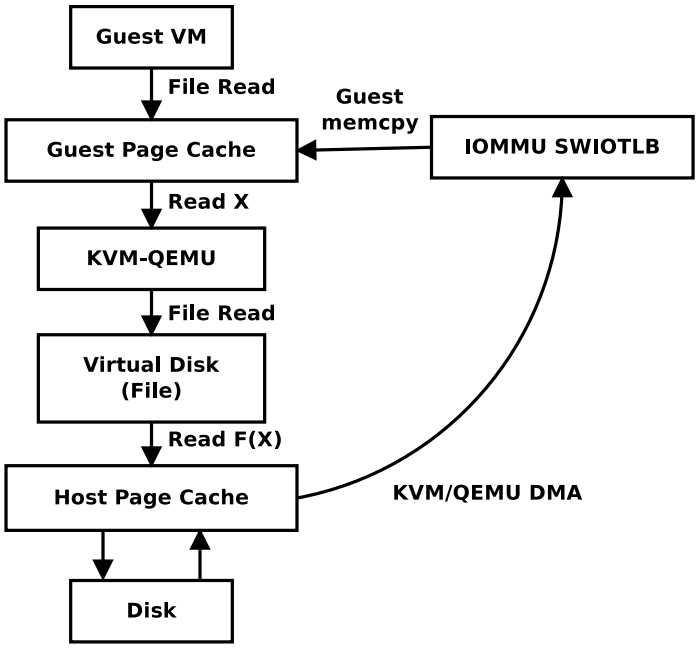


Figure 3.1: An example of a disk I/O operation by an SEV-enabled VM.

3.2 Threat Model

We consider a scenario in which the VMs' memory are encrypted and protected by AMD SEV technology. The hypervisor run on a machine controlled by a third-party service provider. Under the threat model we consider, the third-party service is not trusted to respect

the integrity or confidentiality of the computation inside the VMs. This could happen when the service provider is dishonest or when the hypervisor has been compromised.

The goal of the attacks in this chapter is either to compromise the I/O operations themselves or the memory encryption of SEV. Out of scope in this proposal are denial-of-service (DoS) attacks, in which the service provider simply refuses to run the VM. SEV is not designed to prevent DoS attacks.

3.3 Security issues of Unprotected I/O Operations in AMD SEV

In this section, we explore the security issues of the lack of protection for SEV's I/O operations. We start of our exploration with the most straightforward consequence of vulnerability—the insecurity of I/O operations itself—and present an attack example that breaches the integrity of I/O operations. To comprehensively study the attack surface, we also enumerate the I/O operations from a guest VM that are vulnerable to such attacks and discuss the challenges of implementing effective countermeasures. Next, we show that I/O insecurity leads to a complete compromise of the memory encryption scheme of SEV, by constructing powerful attack primitives that leverage the unprotected I/O operations to enable the adversary to encrypt or decrypt arbitrary messages with the guest VM's memory encryption key, k_{vek}.

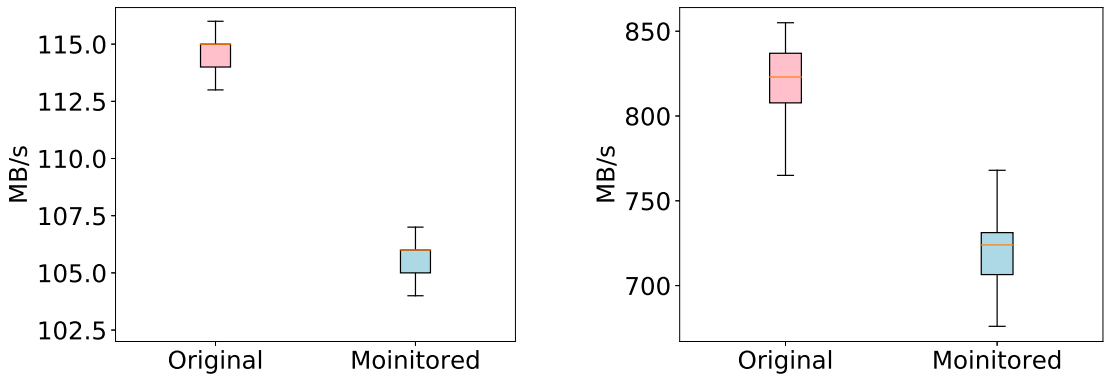
3.3.1 Unprotected I/O Security's consequences

In this section, we explore the direct consequences of unprotected I/O operations from SEV-enabled guests.

3.3.1.1 Case Study: Integrity Breaches of Disk I/O

We first present a case study to show how SEV's guest VMs' unprotected I/O operations can be exploited to breach I/O security in practice. In this case study, we show that a malicious hypervisor is able to gain control of the guest VMs through an OpenSSH server without passwords by exploiting unprotected disk I/O. Therefore, we assume the disk is *not* encrypted with disk encryption key in this example. However, we note it is recommended by AMD to only use encrypted storage. As such, this case study only serves the purpose of proof-of-concept, rather than a practical attack. We will discuss its security implications in Section 3.3.1.2.

Specifically, the adversary controls the entire host and launches the SEV-enabled VM using the standard procedure [7]. During the system bootup, the binary code of `sshd` that performs user authentication is loaded into the memory. To monitor the disk I/O streams, whenever the QEMU performs a DMA operation for the guest, the adversary checks the memory buffer used for this DMA operation (*i.e.*, `SWIOTLB`) and search for the binary code of `sshd`. In our implementation, we used a 32-byte memory content (*i.e.*, 0*x*ff85 0*x*c041 0*x*89c4 0*x*8905 0*x*4e05 0*x*2900 0*x*0f85 0*x*1b01 0*x*0000 0*x*488b 0*x*3d49 0*x*0529 0*x*0089 0*x*eee8 0*x*c2bf 0*x*fdff) as the signature of the `sshd` binary and no false detection was observed. Once the DMA operation for `sshd` is identified, the adversary modifies the binary code inside `SWIOTLB`, before the QEMU commits the DMA operation. In particular, this is done by replacing the crucial code used in authentication that corresponds to `callq pam_authenticate`, which is a five-byte binary string 0*x*e8 0*x*c2 0*x*bf 0*x*fd 0*x*ff, to `mov $0 %eax` (a binary string of 0*x*b8 0*x*00 0*x*00 0*x*00 0*x*00). `pam_authenticate()` is used to perform user authentication; only when it returns 0 will the authentication succeed. Therefore, by moving 0 to the register `%eax` (the register used to store return value of a



(a) I/O write

(b) I/O read

Figure 3.2: Read/write performance overhead due to I/O monitoring.

function call) directly, the adversary can successfully bypass the user authentication without knowing the password. To validate the attack, we empirically conducted the attack three times and all were successful.

Performance degradation due to I/O monitoring. We also conducted experiments to measure the performance degradation due to the hypervisor's monitoring of disk I/O streams. We used the `dd` command to write 1GB of data to the local disk to measure the I/O write speed. The `dsync` flag of set to make sure the data is written to the disk directly, bypassing the page caches. To measure the read speed, we cleaned the page caches in the memory by setting `vm.drop_caches=3` before reading 1GB of data from local disk. In both the read and write experiments, we measured the performance with and without I/O stream monitoring and repeated the measurements 200 times. The results show the performance degradation of I/O read and write is 11.8% and 7.9% respectively (see Figure 6.2).

3.3.1.2 Estimating The Attack Surface

As shown in the above example, I/O operations that are not encrypted by the software can be intercepted by the malicious hypervisor and manipulated to compromise the SEV-enabled guests. This vulnerability exists in all emulated I/O devices that are commonly used in cloud VMs, such as disk I/O, network I/O, and display I/O, *etc.*. While a straightforward solution is to encrypt I/O streams by software, however, this simple method has many practical limitations in practice:

Network I/O. Network traffic can only be partially encrypted, as headers of IP or TCP cannot be encrypted. The adversary is still able to modify the network traffic to forge the IP addresses, port numbers, and encrypted metadata of the network packets. This is true for both TLS traffic and VPN traffic. As we will show in Section 3.3.2, encrypted traffic like SSH can still be exploited to construct memory decryption oracles.

Display I/O. Encrypting I/O traffic cannot be applied when the I/O devices cannot decrypt the I/O stream by themselves. Display I/O is one such example. For instance, Virtual Network Computing (VNC) is a graphical desktop sharing protocol that allows VMs to be remotely controlled. In KVM, the QEMU redirects the VGA display from the guest to the VNC protocol, which is not encrypted. Therefore, if the user of the guest VM uses VNC to control the VM, keystroke and mouse clicking will be learned and manipulated by the adversary. To protect display I/O operations, the guest VM must be modified to encrypt all display I/O traffic and the remote user interface must be modified accordingly to decrypt the traffic.

Disk I/O. For disk I/O operations, the method recommended by SEV [5] is for each SEV-enabled VMs to use encrypted disk filesystems. To use encrypted disks, however, the

owner needs to first provision the disk encryption key into the protected VMs by using the `Launch_Secret` [7] command. This command first decrypts a packet sent by the VM owner (that contains the disk encryption key) encrypted using K_{tek} (Transport Encryption Key), atomically re-encrypts it using the memory encryption K_{vek}, and then injects it into the guest physical address specified by `GUEST_PADDR` (a parameter of the `Launch_Secret` command). As the address of the disk encryption key is known, if memory confidentiality is compromised (using methods to be described in Section 3.3.2), the disk encryption key can be learned and used to decrypt the entire image. Therefore, disk I/O is not secure, either.

3.3.2 Decryption Oracles

In this section, we show that the DMA operations under SEV's memory encryption technology can be exploited to construct a decryption oracle, which allows the adversary to decrypt any memory block encrypted with the guest VMs' memory encryption key K_{vek}. The oracle can be frequently and repeatedly queried and thus can be exploited as an attack primitive for more advanced attacks against SEV-enabled guests.

As mentioned in Section 3.1, the DMA operation from the SEV-enabled VM is conducted with the help of memory pages shared with the hypervisor. When DMA operates in the `DMA_TO_DEVICE` mode, data is transferred by the IOMMU hardware to the shared memory, and then copied by CPU in the SEV-enabled VM to its private memory; when DMA operates in the `DMA_BIDIRECTIONAL` mode, the SEV-enabled VM first copies the data from encrypted memory to the shared memory, and then the DMA reads or writes are performed on the shared memory.

Both these modes of operations provide the adversary an opportunity to observe the transfer of data blocks from memory pages encrypted by K_{vek} to memory pages that is

not encrypted (from the hypervisor's perspective). Therefore, if the adversary alters the ciphertext of the data blocks in the encrypted memory page before they are copied by the guest VM, after the memory copy, the corresponding plaintext can be learned from the shared memory directly.

The construction of such a decryption oracle is shown in Figure 3.3. The decryption oracle can be constructed in three steps: *pattern matching*, *ciphertext replacement*, and *packets recovery*. We use network I/O as an example. The adversary exploits the network traffic in Secure Shell (SSH) to construct the decryption oracle. But we stress that any I/O traffic can be exploited in similar manners. In the following experiments, we configured the guest VM to use `OpenSSH_7.6p1` with `OpenSSL` 1.0.2n, which is default on Ubuntu 18.04.

3.3.2.1 SSH and Network Stacks

To control the SEV-enabled guest remotely, the owner of the VM typically uses SSH protocol to remotely login into the VM and controls its activities. To copy data to and from the VM, protocols like SCP, which is built on top of SSH, is commonly used. Particularly, we consider the SSH traffic after the remote owner has already authenticated with `sshd` and a secure communication channel has been established. Because the SSH handshake protocol is performed in plaintext, the adversary who controls the hypervisor and QEMU can act as a man-in-the-middle attacker and recognize the established the secure channel by its IP addresses and TCP port number. Once the secure channel is established, SSH command and output data will be transferred using encrypted SSH packets that are transmitted in interactive mode [84].

In the interactive mode, each individual keystroke guest owner types will generate a packet that is sent to the SEV-enabled VM, which will be transferred by DMA to a memory buffer shared between the guest and the hypervisor. The packet is then copied by the guest to

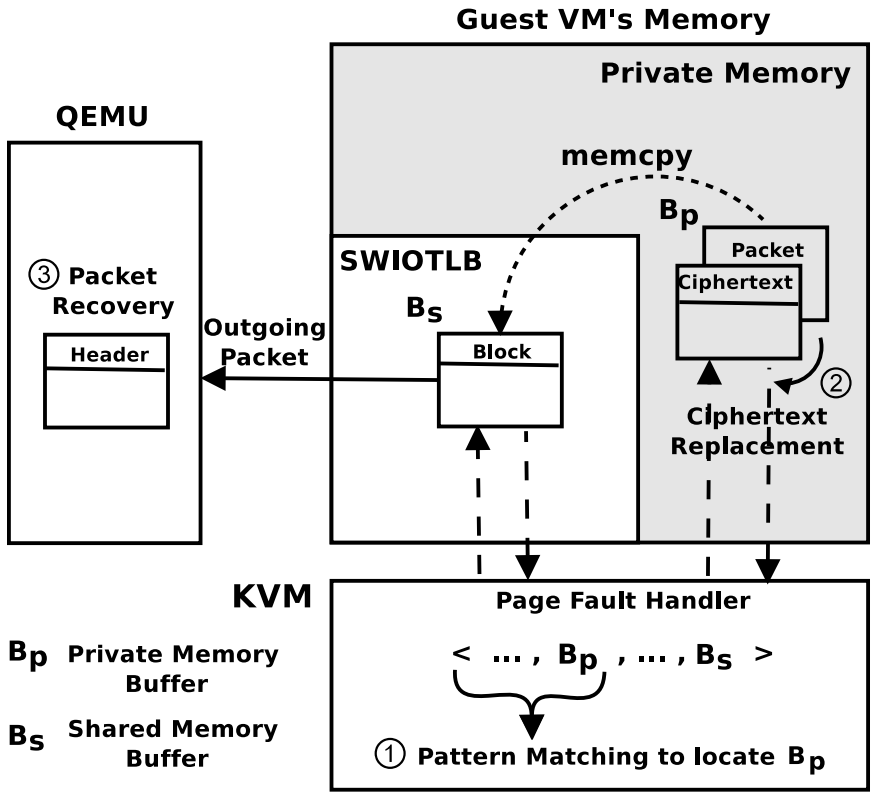


Figure 3.3: A decryption oracle. Step ①, the hypervisor conducts pattern matching using page-fault side channels to determine the address of B_p. Step ②, the hypervisor replaces a ciphertext block in B_p with the target memory block, which will be decrypted when copied to B_s. Step ③, QEMU recovers the network packet headers.

a private memory page encrypted using K_{vek}. Then the data is handled by the network stack in the guest OS kernel. The headers of the packet are then removed and the payload data is forwarded to the user-space application. Then the SSH server processes the keystroke and responds with an acknowledgement packet. The acknowledgement packet is copied back to the kernel space, wrapped by the corresponding header information, and then copied to the shared memory buffer. The last memory copying also decrypts the memory using the guest VM's K_{vek}. Therefore, our attack primitives target this process. As a result, every network

packet generated by the guest VM can be exploited as a decryption oracle that helps the adversary decrypt one or multiple memory blocks.

3.3.2.2 Pattern Matching Using Fine-grained Page-fault Side Channels

Let us denote the private memory buffer as B_p, whose gPA is P_{priv}, and the shared memory buffer as B_s, whose gPA is P_{share}. The primary challenge in this attack is to identify the P_{priv}. As this address is never directly leaked, the adversary needs to perform a page-fault side-channel analysis.

Fine-grained page-fault side channels in SEV. The page fault side channel was first studied by Xu *et al.* in the context of Intel SGX [95]. As an SGX attacker controls the entire operating system, he or she can manipulate the page table entries (PTE) and set the `present` bit of the PTEs of pages that are mapped to the targeted enclave. By doing so, once the enclave program accesses the corresponding memory pages, the control flow will be trapped into the OS kernel through a page fault exception. On x86 processors, the faulting address will be stored in a control register, `CR2` so that the page-fault handler could learn the entire faulting address. To provide secrecy, SGX masks the page offset of the faulting address and leaves only the virtual page number in `CR2`.

Similarly, on the AMD platform, the adversary that compromises the hypervisor could also exploit the page-fault side channels to track the execution of the SEV-enabled VMs. Although the mapping between the guest VM's guest virtual address (gVA) to gPA is maintained by the guest VM's page table and is encrypted by K_{vek}, the hypervisor could manipulate the nested page tables (NPT) to trap the translation from gPAs to host physical addresss (hPA). Unlike SGX, SEV does not mask the page offset, providing more fine-grained observation to the adversary.

Moreover, the page-fault error code returned in the `EXITINTINFO` field of VMCB can also be exploited in the SEV page-fault side-channel analysis. Specifically, the page-fault error code is a 5-bit value, revealing the information of the page fault. For example, when bit 0 is cleared, the page fault is caused by non-present pages; when bit 1 is set, the page fault is caused by a memory write; when bit 2 is cleared, the page fault takes place in the kernel mode; when bit 3 is set, the fault is generated form a `reserved` bit; when bit 4 is set, the fault is generated by an instruction fetch. The error code provides detailed information regarding the reasons of the page fault, which can be leveraged in side-channel analysis.

Pattern matching. With such a fine-grained side channel, the adversary could monitor the memory access pattern of the guest when it receives an SSH packet. Particularly, after delivering an SSH packet to the SEV-enabled VM, the adversary immediately initiates the monitoring process and marks all of the guest VM's memory pages inaccessible by clearing the `present` bit of the PTEs. Every time a memory page is accessed by the guest, a page fault takes place and the adversary is able to learn the entire faulting address P_i. Note here the faulting address in the guest VM refers to the guest physical address as the guest virtual address is not observable by the hypervisor. After the page fault, the adversary resets the `present` bit in the PTE to allow future accesses to the page. Therefore, with the fine-grained page fault side channel, one only needs to collect information regarding the first access to a memory page. The monitoring procedure stops when the acknowledgement packet is copied into B_s. At this point, the adversary has collected a sequence of faulting addresses $<P_1, P_2, \cdots, P_m>$.

Internally in the guest VM, when *sshd* is sending a packet, the encrypted data is first copied to the buffer of the transport layer, then the buffer of the network layer, and then the buffer of the data link layer. In each layer, new packet headers are added. Eventually, the

entire network packet is stored in a data structure called `sk_buff`. Finally, the kernel will call `dev->hard_start_xmit` to transfer the data in `sk_buff` to the device driver, where B_p is located.

Both P_{priv} and the address of `sk_buff`, P_{sk}, should be found in the faulting addresses sequence $<P_1, P_2, \cdots, P_m>$. It is because the memory pages that store the private memory buffer B_p and `sk_buff` are not otherwise used during the process of sending network packets. The adversary could combine page offsets, page frame numbers, the page-fault error code, and the number of page faults between the two page faults of B_p and `sk_buff` to create a signature, which can be used to find P_{priv}. For example, the page-fault error code of B_p is $0b110$ and the page-fault error code of `sk_buff` is $0b100$; the page offset of P_{priv} is usually $0x0$fa or $0x8$fa and the offset of `sk_buff` usually ends with $0xe8$ or $0x00$; and the number of page faults between B_p and `sk_buff` is roughly 20. With these signatures, the adversary can identify P_{priv} from the sequence of faulting addresses. Of course, the signature may change from one OS version to another, or change with different OS kernel. However, because the adversary controls the hypervisor, such information can be re-trained offline, before performing the attacks.

It was indicated by AMD researchers (during an offline discussion) that SEV-ES should mask the page offset information when there is a VMEXIT. However, we were not able to find related public documentation. Moreover, as the KVM patch for SEV-ES support is not yet available at the time of writing, we were not able to validate the claim or estimate the remaining leakage (*e.g.*, error code, page offset) after the patch. However, regardless of the hardware changes, a coarse-grained page-fault side channel in which the page frame number of the faulting address is leaked must remain. To show that the demonstrated attack still works, we conducted experiments to perform pattern matching without page fault offsets

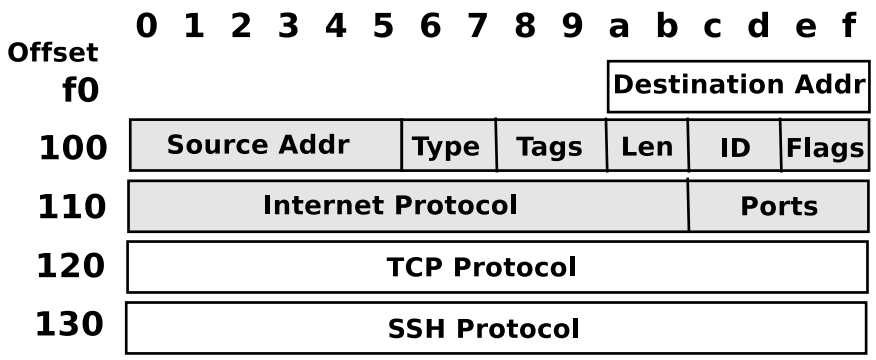


Figure 3.4: Format of an SSH packet.

and error code information. Specifically, we performed pattern matching using only the faulting page numbers, with the guest VM running different Ubuntu versions (*e.g.*, 18.04, 18.04.1 and 19.04) and different kernel versions (4.15.0-20-generic, 4.15.0-48-generic and 5.0.0-13-generic). The results show that after training in one virtual machine, the pattern matching rules can work well even in different virtual machines with the same Ubuntu version and kernel version—the attacker is still able to successfully identify the page frame number of P_{priv}. To determine the complete address of P_{priv}, the attacker could determine the offset by scanning the entire memory page and looking for content changes (*e.g.*, in a 90-byte buffer).

3.3.2.3 Replacing Ciphertext

After determining P_{priv}, the adversary replaces aligned SSH header in B_p with the ciphertext he or she chooses to decrypt. As shown in Figure 3.4, the packet headers include a 6-byte destination address, a 6-byte source address, a 2-byte IP type (*e.g.*, IPv4 or IPv6), 1-byte IP version and IP header length, 1-byte of differentiated services field, 2-byte packet length, 2-byte identification, 2-byte of IP flags, 1-byte time-to-live, 1-byte protocol type,

2-byte checksum, and 4-byte source IP address and 4-byte destination address, and 20-bytes TCP headers (start with 2-byte source port and 2-byte destination port).

As shown in Figure 3.4, P_{priv} has the offset address ending with $0xfa$. Because SEV encrypts data in 16-byte aligned blocks, only part of the TCP/IP header (i.e., header in gray blocks in Figure 3.4) can be used to decrypt ciphertext. Additional constraints apply if the packet needs to be recovered later. Before replacing the packet header with the chosen ciphertext, the adversary performs a `WBINVD` instruction to flush the guest VM's cached copy of B_p back to memory. It is because cache coherence is not maintained by the hardware between cache lines with different ASIDs. To make sure the guest VM's copy does not overwrite our changes to the memory, `WBINVD` instruction needs to be called first.

The ciphertext replacement takes place before `memcpy`, after B_p is accessed and before B_s is accessed. B_s is located inside the `SWIOTLB` pool, which is the next available address within `SWIOTLB` that can be used by the guest. After replacing a few blocks in B_p, another `WBINVD` instruction is performed to ensure the guest VM reads and decrypts up-to-date ciphertext in memory. All replacement operation is achieved by `IOremap` instead of `Kmap`, since `Kmap` decrypts data with the hypervisor's key first and `IOremap` directly operates data in the memory without decryption.

We use the following example to illustrate the attack. Let the ciphertext c be a 16-byte aligned memory block with the gPA of P_c. The function which can translate gPA to hPA is called $hPA()$. The goal of the attack is to decrypt c. The adversary replaces a 16-byte data in the SSH header that begins with address $(P_{priv}+16)/16*16$ with c. After the data in B_p is copied to B_s, the adversary could read the decrypted SSH packet and extract the plaintext of decrypted memory block, d, from the corresponding location of the packet. However, d is not the plaintext of c yet, as SEV's memory encryption involves a tweak function $T()$. That is,

$c = E_{K_{vek}}(m \oplus T(hPA(P_c)))$ but $d = D_{K_{vek}}(c) \oplus T(hPA((P_{priv}+16))/16*16)$. Therefore, the plaintext message m of ciphertext c can be calculated by $m = d \oplus T(hPA((P_{priv}+16))/16*16) \oplus T(hPA(P_c))$.

3.3.2.4 Packets Recovery

To make the attack stealthy, the adversary needs to recover the network packet with decrypted data before those packets are passed to the physical NIC device. As shown in Figure 3.4, the SSH header also contains metadata of the packet. When the malicious hypervisor injects chosen ciphertext into the memory block with offset = $0x100$, the adversary only needs to be concerned about a portion of the source IP address, IP protocol type, IP tags, TCP header length, and the identification of the packet. Majority of the fields are determined. The identification of the IP packet increases by 1 every time SSH server replies a packet. So when hypervisor tries to recovery the (plaintext) packet from the QEMU side, it only need to correct the packet length, increase identification by 1 and copy the remaining portion from previous packet such as source address, header length, time to live and protocol number.

3.3.3 Encryption Oracle

We next show the construction of a memory encryption oracle using unprotected I/O operations. The encryption oracle stealthily encrypts a chosen plaintext message using a guest VM's memory encryption key K_{vek}. Similar to the construction of the decryption oracle, during the DMA operation of the guest that transfers data from the device to the encrypted memory, the adversary changes the message m in the shared memory buffer B_s, waits until it is copied to the private buffer B_p in the encrypted page, and then extracts the corresponding ciphertext $E_{k_g}(m)$ from B_p.

To determine the gPA address of B_p and retrieve the ciphertext of the plaintext message at address P_t, the steps shown in Figure 3.5 are taken. Again, we leverage the fine-grained page-fault side channel we used in the previous section. Specifically, we modified all memory pages' PTEs right after the QEMU finishes writing the packet into `SWIOTLB` and before the QEMU notifying guest VM about the DMA write. Then, when the guest VM performs a `memcpy` operation to copy the data, the adversary will observe a sequence of page faults: <...P_{share}, P_{priv}...>, where P_{share} is the address of B_s and P_{priv} is the address of B_p. The page fault at P_{priv} will take place right after the page fault at P_{share}. When the hypervisor handles the page fault at P_{priv}, it replaces the 16-byte aligned data block with the message m', where $m' = m \oplus T(hPA(P_{priv})) \oplus T(hPA(P_t))$, where P_t is the gPA of the target address to which the adversary wishes to copy m. The corresponding ciphertext will be $c = E_{k_g}(m \oplus T(hPA(P_t)))$, which can be used to replace the ciphertext at address P_t.

The encryption oracle can be typically exploited to inject code or data into the SEV-enabled VM's encrypted memory, or it can be used to make guesses of the memory content by providing a probable plaintext. We note that to use the encryption oracle, the adversary may simply generate meaningless packets and send them to the guest VM, which will be discarded. But the oracle can still be constructed and used. The only downside of this approach is that the guest VM will observe large volume of meaningless network traffic and may become suspicious of attacks.

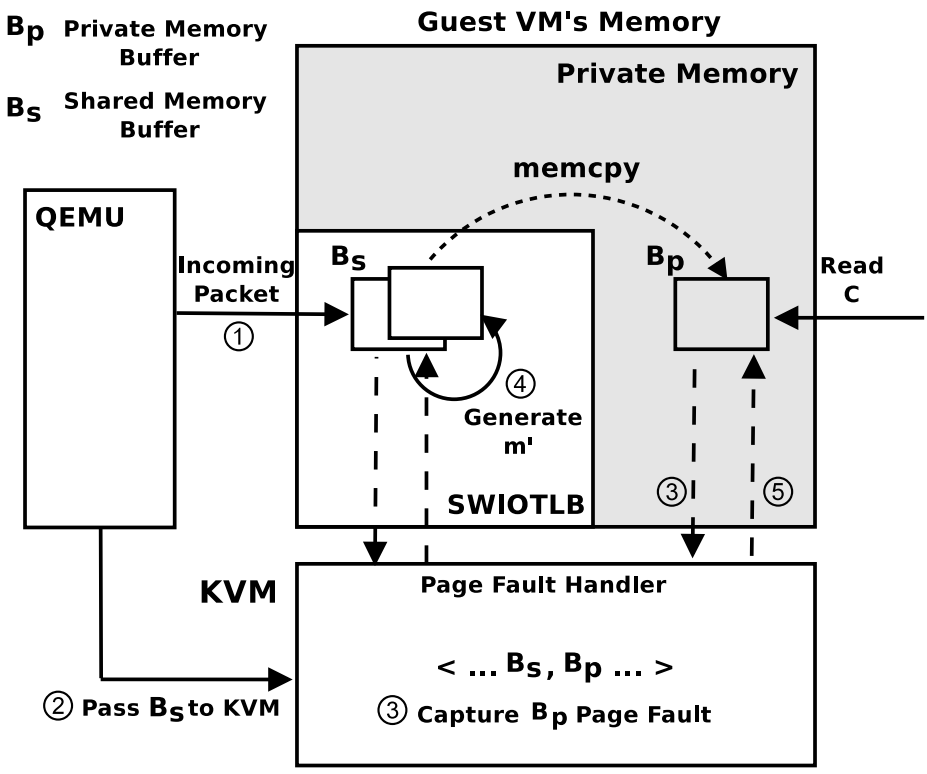


Figure 3.5: An encryption oracle. Step ①, QEMU forwards an incoming packet to the guest. Step ②, QEMU passes the address of B_s to the hypervisor. Step ③, a page fault immediately after the fault at B_s is captured by the page fault handler. Step ④, message m' is placed in B_p. Step ⑤, page fault handler returns the control to the guest.

3.4 Evaluation

We implemented our attacks on a blade server with an 8-Core AMD EPYC 7251 Processor, which has SEV enabled on the chipset. The host OS runs Ubuntu 64-bit 18.04 with Linux kernel v4.17 (KVM hardware-assisted virtualization supported since v4.16) and the guest OS also runs Ubuntu 64-bit 18.04 with Linux kernel v4.15 (SEV supported since v4.15). The QEMU version used was QEMU 2.12. The SEV-enabled guest VMs were configured with 1 virtual CPU, 30GB disk storage, and 2GB DRAM. The OpenSSH server was installed from the default package archives.

3.4.1 Pattern Matching

We first evaluate the pattern matching algorithm's accuracy of determining P_{priv}. To obtain the ground truth, we modified the guest kernel to log the gPA address of `sk_buff`, the source gPA and destination gPA of `memcpy`, as well as the size of each DMA read or write. All the data was recorded in the kernel debug information, which can be retrieved using a Linux command `dmesg`.

The experiments were conducted as follows: We ran a software program *AnJian* [31] (an automated keystroke generation tool) on a remote machine, which opened a terminal that was remotely connected to the SEV-enable VM through an SSH communication channel. *AnJian* automatically typed on the SSH terminal two Linux commands `cat security.txt |grep sev` and `dmesg` at the rate of 10 keystrokes per second. This was used to simulate the remote owner controlling the SEV-enabled VM through SSH. The adversary would make use of the generated SSH packets to perform memory decryption. The `dmesg` command also retrieved the kernel debug message that recorded the ground truth.

At the same time, the pattern matching was performed by the adversary on the hypervisor side. The page-fault side-channel analysis was conducted upon receiving every incoming SSH packet to guess the address P_{priv}. There were three outcomes of the guesses: a correct guess, an incorrect guess, and unable to make a guess. Because there were 33 keystrokes generated by *AnJian*, the adversary was allowed to guess P_{priv} for 33 times in each experiment. The experiments were conducted 20 times.

Figure 3.6 shows the precision and recall of these 20 rounds of experiments. Precision is defined as the ratio of the number of correct guesses and the number of times that a guess can be made. Recall is defined as the ratio of the number of correct guesses and the number

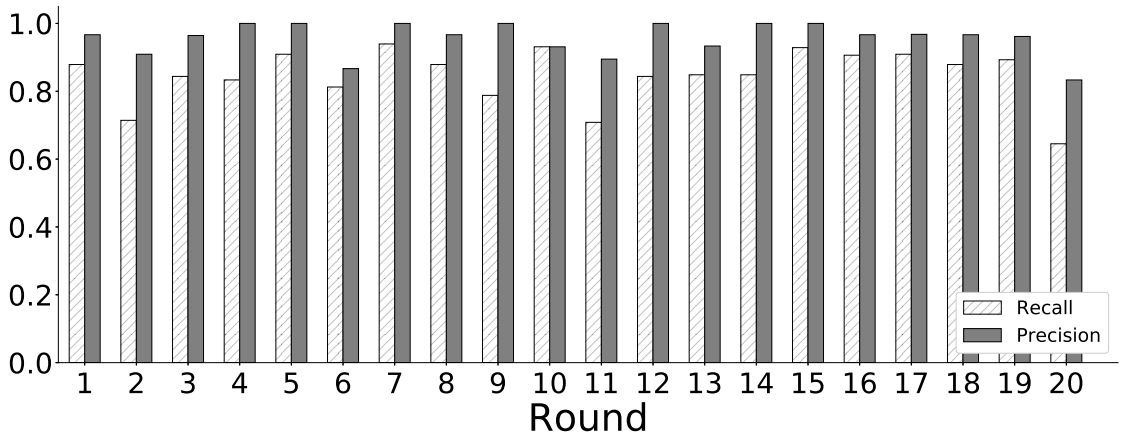


Figure 3.6: The precision and recall of determining P_{priv} in 20 rounds or experiments.

of total SSH packets. The average precision is 0.956, the average recall is 0.847 and the average F_1 Score is 0.897.

3.4.2 Persistent B_p

According to our experiments, the B_p will remain unchanged and reused for multiple network packets. This greatly helps the adversary, either by performing pattern matching once and reusing the same B_p directly in subsequent packets, or by improving the accuracy of the guesses.

Improving attack fidelity using persistent B_p. The persistent B_p can be used to reduce the number of incorrect guesses. During a real-world attack, when P_{priv} is incorrectly guessed, the ciphertext replacement may crash the guest VM (although we have not experienced any crashes in our experiments). As such, a safer strategy of when to perform ciphertext replacement is only after correctly guessing P_{priv} N times in a streak, which we call the *N-streak strategy*. We then applied this strategy to Round 20, 6 and 11, which have the highest FPR (*i.e.*, 0.167, 0.133, 0.103, respectively). As shown in Figure 3.7, when by

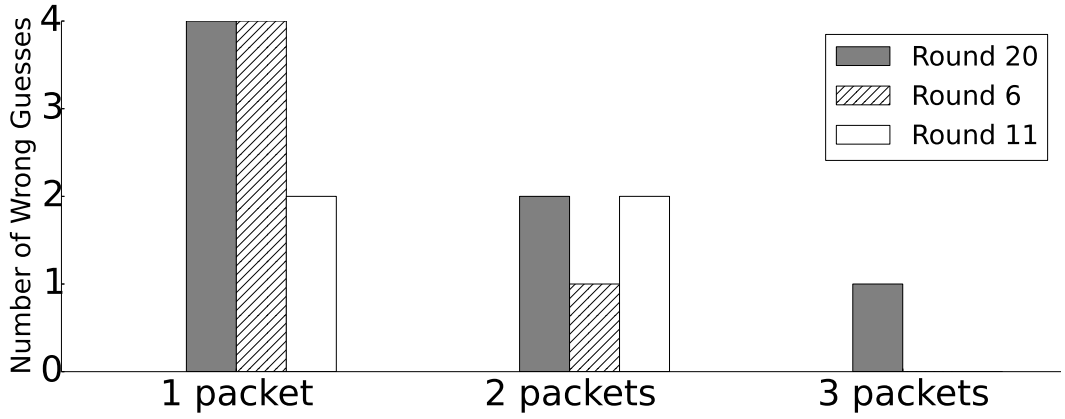


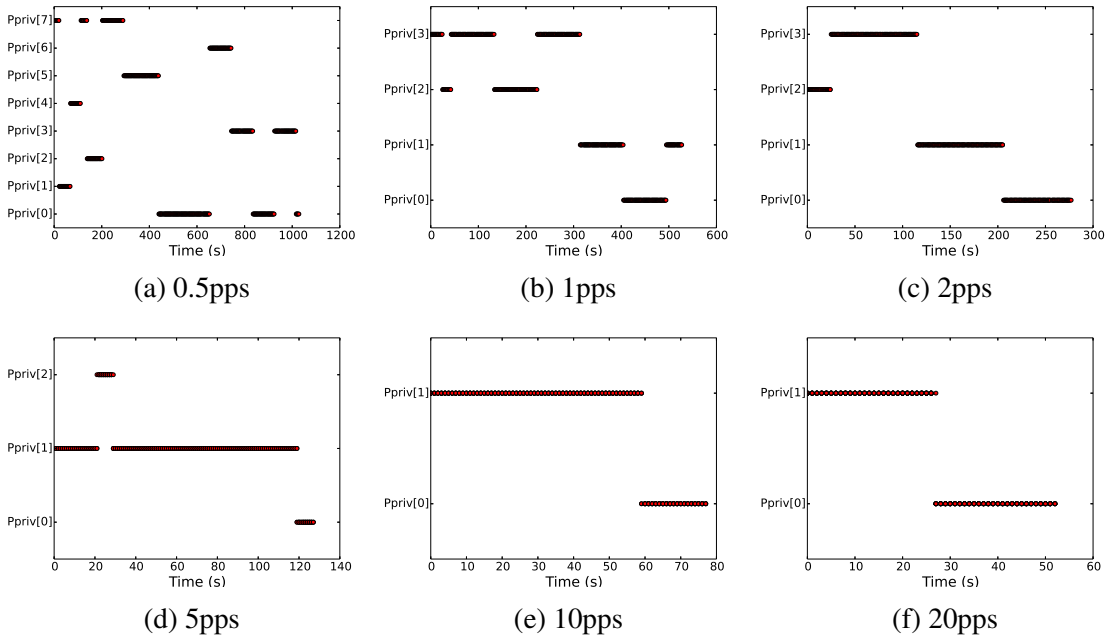
Figure 3.7: Reduction of incorrect guesses using the N-Streak strategy.

increasing N (*i.e.*, 1, 2, 3), the number of incorrectly performed ciphertext replacement is reduced.

Packet rate vs. B_p persistence. We further evaluated the effect of B_p persistence when the rate of SSH packets varies. Again, on the remote machine, we used *AnJian* to generate keystrokes at a fixed rate, ranging from 0.5 keystrokes per second, to 20 keystrokes per second. The rate of SSH acknowledgement packets is close to the keystroke rate. For each keystroke rate, 500 keystrokes were generated and the number of different B_ps were reported in Figure 3.8. We can see that as the packet rate increases, fewer number of B_ps will be used to send SSH packets. We repeated this experiment and collected over 200 different B_ps after generating 5000 keystrokes with rates ranging from 0.5 to 20 per second. The statistics of the repeated use of B_ps are shown in Figure 3.9.

3.4.3 I/O Performance Degradation

Conducting page-fault based side-channel analysis to guess P_{priv} and performing ciphertext replacement will slow down the I/O operations of the guest VM. To evaluate the degree



(a) 0.5pps (b) 1pps (c) 2pps

(d) 5pps (e) 10pps (f) 20pps

Figure 3.8: The number of different B_ps used with various rates of packets (pps).

of performance degradation, we evaluate the SSH response time on the server side during the attacks. The SSH response time measures the time interval between the QEMU receives an incoming SSH packet to the time that an SSH response packet is sent to QEMU. Note the measurements do not include network latency.

Figure 3.10 shows the SSH response time under three conditions: Original (not under attack), B_p Persistent (assuming B_p does not change) , and Guess Every Time (assuming B_p changes and making guesses every time). The keystroke rate used in the experiments were 10 keystrokes per second, and in total $1,000$ keystrokes were generated during the tests. We can see from the figure, the average SSH response latency without attack is 2.5ms and the median is 0.99ms. The average latency for SSH connection under a B_p-persistent strategy is 6.81ms and the median is 2.4ms. The average latency for SSH connection under

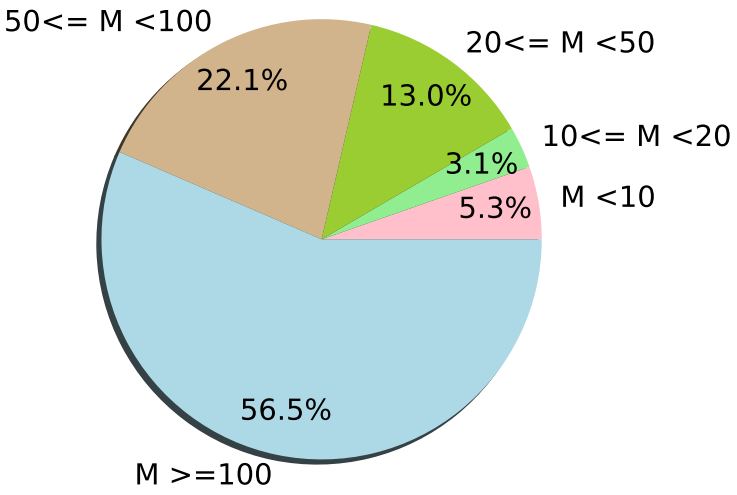


Figure 3.9: Statistics of repeated B_ps.

a guess-every-time strategy is 8.0ms and the median is 8.7ms. Because the typical network latency of cloud servers are 40-60ms within US and more than 100ms worldwide [15], it is very difficult for the VM owners to detect the latency caused by the attacks.

3.4.4 An End-to-End Attack

We conducted an end-to-end attack in which the adversary decrypts a 4KB memory page that is encrypted with the guest VM's K_{vek}. The attack assumes a network traffic with the rate of 10 pps, which is simulated using the same method used in the previous sections. Table 3.1 shows the number of packets and time used to complete the attack, when one or two 16-byte aligned blocks were exploited for the data decryption. We can see that in the four trials we conducted, roughly 300 packets are needed to decrypt the 4KB page, which takes about 40 seconds. The speed of the attack doubles if the first two blocks of the packets were used to decrypt data.

Table 3.1: End-to-end attack performance.

Round	1 Block		2 Blocks	
	Packets used	Time(s)	Packets used	Time(s)
1	292	43.56	148	21.29
2	329	40.78	177	20.04
3	326	39.21	154	18.99
4	299	33.58	154	16.95

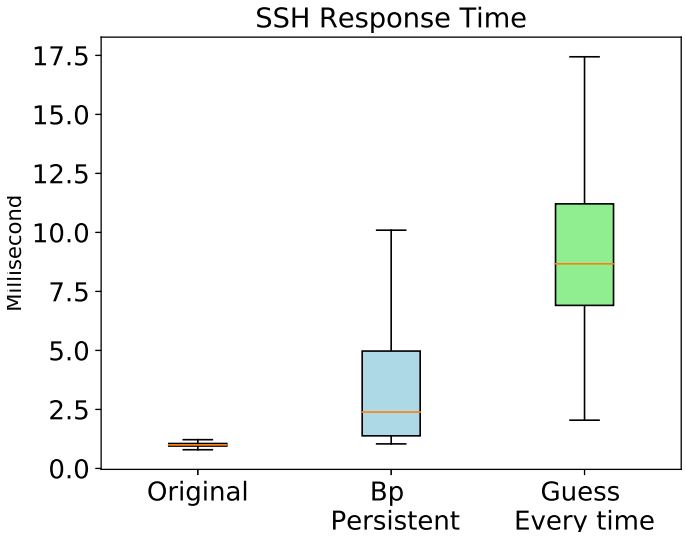


Figure 3.10: I/O performance degradation evaluated using SSH response time (network latency excluded).

3.5 A Path Towards I/O Security in SEV

In this section, we will discuss some potential solution to I/O security in SEV from both hardware and software level. The root cause of the problem is the incompatibility between AMD-V's I/O virtualization with SEV's memory encryption scheme. Specifically, the primary reason of the attacks described in Section 3.3 is that existing IOMMU hardware only supports memory encryption with ASID = 0 and the operated memory is encrypted

with the hypervisor's memory encryption key. Therefore, every I/O operation from the guest VM must go through a shared memory page with the hypervisor. To address this limitation, IOMMU must allow DMA operations to be performed under the ASIDs of other contexts. Meanwhile, it must prevent the privileged software from abusing such IOMMU operations. This design, however, will be very challenging to implement in practice. According to our discussion with AMD researchers, future releases of SEV CPUs are *unlikely* to address this issue. Therefore, alternative solutions must be identified.

In addition to this fundamental issue, the decryption oracle is also enabled by two other vulnerabilities of SEV: (1) no integrity protection of the encrypted memory, and (2) knowledge of the tweak function $T()$. AMD researchers suggested that future SEV CPUs will disable the encryption oracle by providing memory integrity and altering the implementation of the tweak function $T()$. While authenticated memory encryption disables all known attacks against SEV, details of its implementation are yet to be disclosed. We discuss some of the potential considerations in Section 3.5.1. Future versions of the tweak function will be implemented as $T(k,a)$, where a is the physical address and k is a random input that changes after every system reboot. We leave the investigation of these vulnerabilities to future work when the technical details are published. In Section 3.5.2, we present a temporary software fix that works on existing AMD processors (Section 3.5.2).

It is worth noting that AMD researchers suggested that SEV-ES masks the page offset during page fault. However, we could not find relevant documentation or validate the claim on our testbed. Nevertheless, our analysis (see Section 3.3.2.2) suggests that the attack is still effective when the page offset information is unavailable. Specifically, we empirically evaluated the attack method that does not rely on page offsets by repeating the experiments

in Section 3.4.1: the mean precision is 0.900, the mean recall is 0.730 and the mean F_1 score is 0.800, which is only slightly lower.

3.5.1 Authenticated Encryption

Authenticated encryption must be adopted to prevent replay attacks and replacement attacks of the encrypted ciphertext. Merkle Tree (MT) [32] has been proposed for detecting replay and replacement attacks for protecting memory integrity. MT can be built and maintained over any region of memory, and hence it can be used to protect the entire memory or only memory allocated to a VM, or any portion of it. There are two types of MT that can be used, depending on the encryption mode. For direct encryption mode, the MT covers data. For counter-mode encryption, it was shown that replay was only possible if the attacker replays data, its hash, and its counter simultaneously. Hence, protecting counter freshness is sufficient to protect against replay [75]. MT over counters is referred to as Bonsai Merkle Tree (BMT), a variant of which was chosen for implementation in Intel SGX MEE [42].

A fundamental trade off exits between the choice of encryption mode and the overheads of MT. When 128-bit hash is used for MT, MT (and hashes) over data incur memory capacity overhead of 33% (i.e. data-to-MT nodes ratio of 2:1), as illustrated in Figure 3.11. On the other hand, BMT incurs an overhead of 20% for hashes, plus 0.4% for BMT nodes. Hashes are needed for both encryption modes to protect against non-replay tampering of memory data. In addition, counter-mode encryption requires additional storage for counters, which depends on the type of counters. 64-bit monolithic counters take up 9% overheads, but split-counters [97] take up only 1%. Taken together, protection against replay incurs

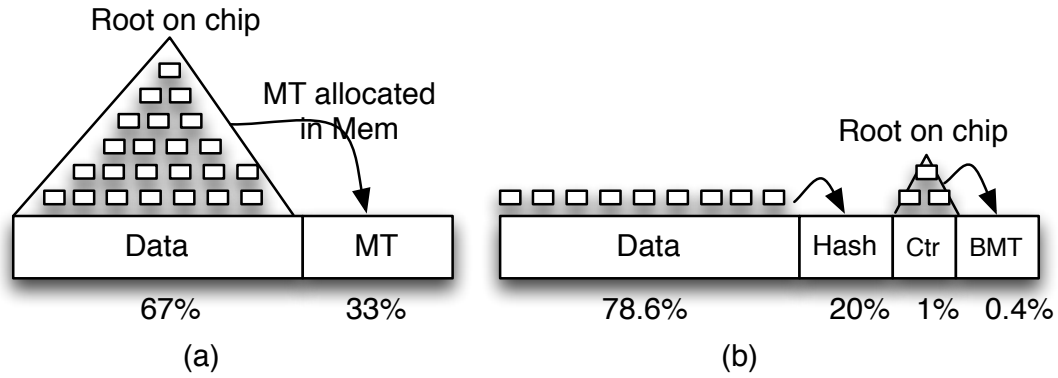


Figure 3.11: Merkle Tree (a) and Bonsai Merkle Tree used in conjunction with split counter mode encryption (b).

13% memory capacity with direct encryption but only 1.4% with counter mode encryption (Figure 3.11). For a 1 TB memory, the difference amounts to a substantial 116 GB.

Counter mode encryption is illustrated in Figure 3.12. Counters are cached on chip, either in regular caches, or in a special "counter cach". With split counters, each page (4KB) of data has its own major counter, and each block (64B) in a page has its own minor counter. When there is a last level cache (LLC) miss, the counter values (major and minor) are concatenated with the page ID and block address of the page (i.e. page offset of the block) to produce a spatially and temporally unique initial vector (IV) [97]. The IV is then encrypted to produce a pad, which will be XOR-ed with ciphertext of data fetched from memory to yield data plaintext. With Galois Counter Mode, the hash of data is obtained a few clock cycles later. Counter mode encryption is more secure than direct mode encryption due to spatial and temporal uniqueness of ciphertexts even for a single plaintext value. Furthermore, as illustrated in the figure, decryption latency is largely overlapped with LLC miss latency; the only exposed latency is 1-cycle XOR of pad and data ciphertext.

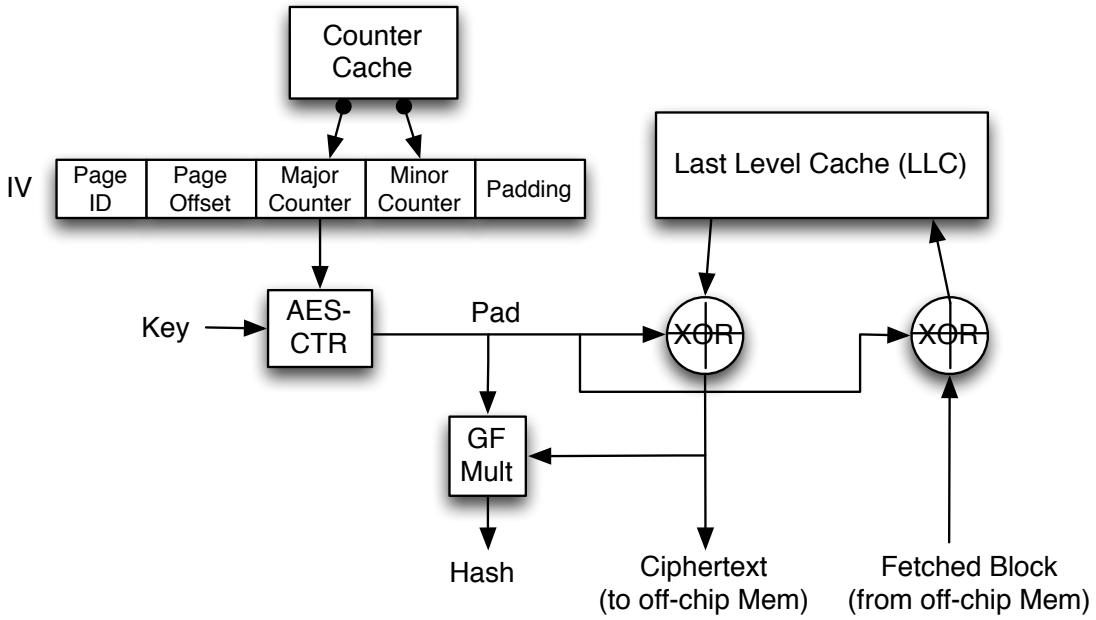


Figure 3.12: Counter mode encryption with split counters using Galois Counter Mode authentication.

In contrast, direct memory mode fully exposes decryption latency in the critical path of memory fetch. Therefore, in terms of security protection against replay, memory capacity costs, and performance, AMD SEV can benefit from counter mode encryption and BMT.

A MT/BMT protects the memory from replay or tampering at all time. It is also possible to selectively protect memory region integrity only at times in which they are "expected" to be vulnerable. For example, the time window in which IOMMU buffer is vulnerable is between the time it is written by the DMA until it is read/consumed by the VM. One could take a hash of the memory region at DMA write and verify it when the VM reads the region. Any tampering or replay attempts will be detected. Selective integrity protection may obviate the need for full MT/BMT for attacks that occur within the vulnerable window, but leaves the memory integrity unprotected at other time.

3.5.2 A Temporary Software Solution

In this section, we present a software solution that can temporarily solve the I/O insecurity issues discussed in this chapter. The key idea is to make sure the hypervisor never observe any unencrypted I/O data to/from the SEV-enabled VM. This can be achieved using SEV's platform management APIs [7] and the transport encryption key of the VM K_{tek}.

K_{tek} is a shared Diffie-Hellman (DH) key between the VM owner and the SEV firmware. Particularly, to launch an SEV-enabled VM on an SEV platform, the owner of the VM first requests the Diffie-Hellman (DH) certificate from the platform, which contains the platform's DH public key. The corresponding private key is kept inside the SEV firmware, which cannot be extracted by the system administrator or the hypervisor. The VM owner then sends her DH public key to SEV platform, so that she establishes a shared transport encryption key K_{tek} with the SEV firmware. K_{tek} is only known by the VM owner and the SEV firmware, but not known to the VM itself or hypervisor. `SEND_UPDATE_DATA` and `RECEIVE_UPDATE_DATA` are two commands (among many others) implemented by SEV to assist the hypervisor to launch and manage SEV-enabled VMs [7]. After the VM is launched, the hypervisor may use SEV's `SEND_UPDATE_DATA` command to *atomically* decrypt a piece of memory with K_{vek} and re-encrypt with K_{tek} or use `RECEIVE_UPDATE_DATA` command to decrypt the memory with K_{tek} and re-encrypt with K_{vek}.

Our proposed solution retrofits these APIs and K_{tek} to protect I/O operations. Particularly, the guest VM kernel and the QEMU can be modified so that the guest VM never copies data between the encrypted memory and the unencrypted memory. Instead, to perform any I/O operation to the SEV-enabled VM, the hypervisor issues the `SEND_UPDATE_DATA` and `RECEIVE_UPDATE_DATA` commands to atomically decrypt and re-encrypt data using the two keys K_{vek} and K_{tek}. As both keys are protected inside the SEV firmware, the hypervisor

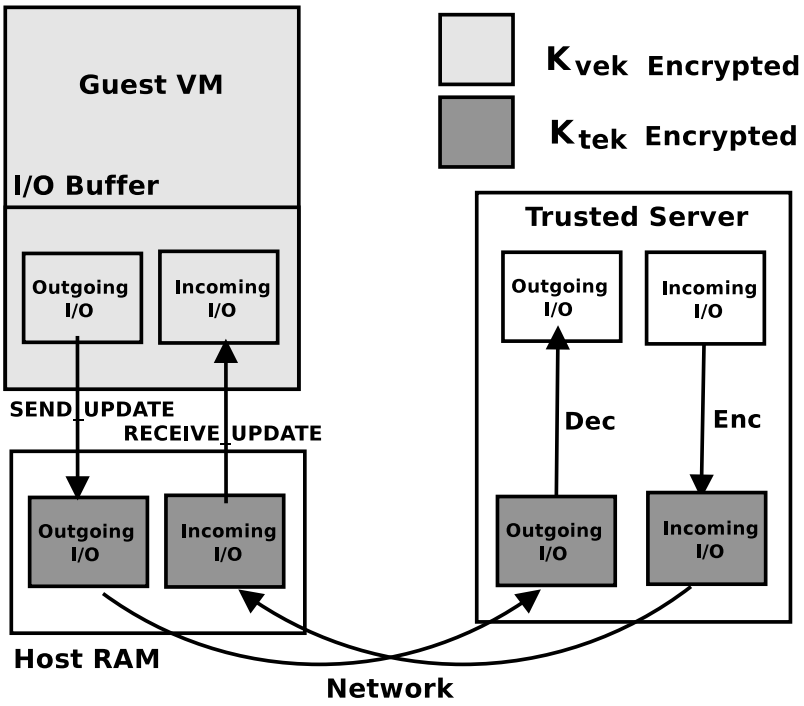


Figure 3.13: An illustration of the temporary software solution.

is not able to learn the plaintext during the I/O operations. The SEV firmware serves as a trusted relay of the I/O paths.

However, this solution is only a temporary fix of the issue. This is because the I/O traffic is encrypted with K_{tek}, which is only known to the owner of the VM. Therefore, all I/O operations, including network I/O, disk I/O, and display I/O must be forwarded to a trusted server that is controlled by the VM owner (as shown in Figure 3.13). Acting as an I/O proxy, the trusted server may limit the application scenarios of SEV and greatly reduce the I/O performance.

3.5.3 Comparison with existing attacks

While the security issues of SEV's I/O operations are orthogonal to the problems of unauthenticated memory encryption, the decryption oracle presented in this chapter does rely on the lack of integrity protection for the ciphertext blocks. However, compared to previous memory decryption attacks against SEV [68, 69], our work differs primarily in three aspects. First, Morbitzer *et al.* [68, 69] manipulate unprotected nested page tables to decouple the mapping between the gVAs and the memory contents, while our decryption oracle directly replaces memory blocks used in the I/O buffer. The hardware mechanisms to defend against these two attacks may differ. Our attack highlights the necessity of mitigating both threats. Second, instead of exploiting a network-facing application executed in the guest VM to accept attacker-controlled data, our attack could make use of any I/O traffic, which is more general. Our work suggests that application-specific defenses, such pruning secrets after use [68], may not work. Third, the attack in Morbitzer *et al.* requires the attacker to actively generate network traffic to the guest VM, which makes it easily detectable. In contrast, our decryption oracle can make use of existing I/O traffic, which can be very stealthy. Moreover, while the memory integrity issues are expected to be addressed in the next release of SEV CPUs, the fundamental I/O security problem studied in this chapter will remain. The encryption oracle will not be mitigated unless the tweak function is completely secured.

3.6 Summary

In this chapter, we have reported our study of the insecurity of SEV from the perspective of the unprotected I/O operations in SEV-enabled VMs. The results of our study are two fold: First, I/O operations from SEV guests are not secure; second, I/O operations can be

used by the adversary to construct memory encryption and decryption oracles. The concrete attacks have been demonstrated in the chapter, along with discussion of potential solutions to the underlying problems.

Chapter 4: Breaking "Security-by-Crash" based Memory Isolation in AMD SEV

In previous chapter, we presented unprotected I/O operations's vulnerabilities in SEV. In this chapter, we fully study SEV's improper use of address space identifier (ASID) for controlling accesses of a VM to encrypted memory pages, cache lines, and TLB entries. We then present the CROSSLINE attacks, a novel class of attacks against SEV that allow the adversary to launch an attacker VM and change its ASID to that of the victim VM to impersonate the victim. We present two variants of CROSSLINE attacks: CROSSLINE V1 decrypts victim's page tables or memory blocks following the format of a page table entry; CROSSLINE V2 constructs encryption and decryption oracles by executing instructions of the victim VM. We have successfully performed CROSSLINE attacks on SEV and SEV-ES processors.

Responsible disclosure. We have disclosed CROSSLINE attacks to AMD via emails in December 2019 and discussed the findings in this chapter with AMD engineers by phone in January 2020. The demonstrated attacks and their novelty have been acknowledged. As discussed in the chapter, neither of the two attack variants directly affect SEV-SNP. Therefore, AMD would not replace ASID-based isolation in the short term, but may invest more principled isolation mechanisms in the future.

The rest of the chapter is organized as follows: Section 4.1 explains SEV's ASID-based isolation for memory, cache and TLB, and explains how ASIDs are managed by the hypervisor. Section 4.2 introduces the two variants of CROSSLINE attacks. Section 4.3 describes the applicability of CROSSLINE on SEV-ES. Section 5.6 presents an extension of CROSSLINE that exploits issues of TLB isolation in SEV VMs, a discussion on CROSSLINE's applicability to SEV-SNP and to Intel processors. Section 4.5 summarize this chapter.

4.1 Demystifying ASID-based Isolation

ASID was initially designed by AMD to tag TLB entries so that unnecessary TLB flushes can be avoided when switching between guest VMs and the host. SEV reuses ASID as the indices of VEKs stored in AMD-SP. Cache tags are also extended accordingly to isolate cache lines with different ASIDs. As a result, ASID becomes the de-facto identifier used by SEV processors to control the software's accesses to virtual memory, caches, and TLBs (as shown in Figure 4.1).

However, following AMD-V, SEV allows the hypervisor to have (almost) complete authority over the management of ASIDs, which gives rise to security concerns as a malicious hypervisor may abuse this capability to breach ASID-based isolation. Interestingly, AMD adopts a "security-by-crash" and assumes if "*the wrong ASID is used for a guest*", the execution of the instruction will "*likely cause a fault*" [51]. In this section, we set off to understand and demystify how ASIDs are used to isolate memory, cache, and TLBs in SEV, and how ASIDs are managed by the hypervisor.

4.1.1 ASID-based Isolation

First, we explore in depth how ASID is used for access control in the virtual memory, CPU caches, and TLBs.

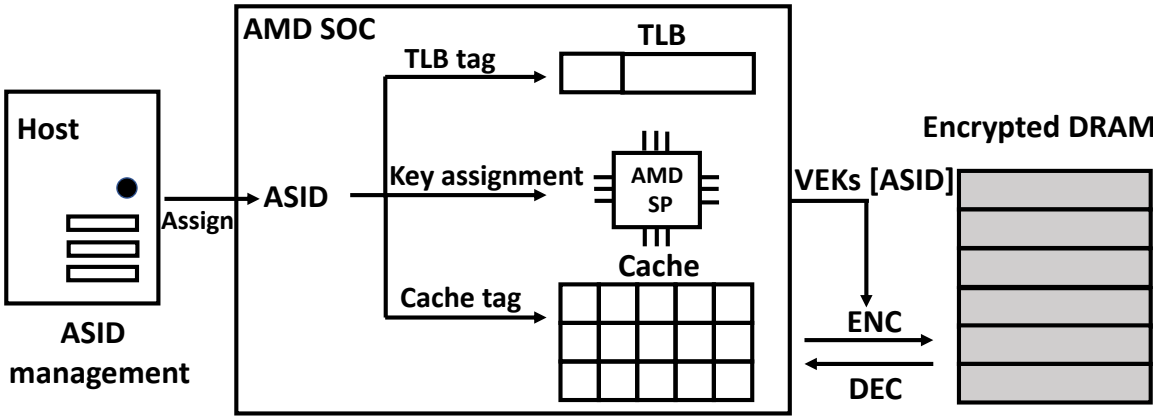


Figure 4.1: ASID-based memory isolation in SEV.

4.1.1.1 ASID-based Memory Isolation

ASIDs are used by the AMD-SP to index VEKs of SEV VMs. The SEV hardware ensures the data and code of an SEV VM is encrypted in the DRAM and only decrypted when loaded into the SOC. Specifically, each memory read from an SEV VM consists of memory fetches by the memory controller of a 128-bit aligned memory block, followed by an AES decryption by AMD-SP using the VEK corresponding to the current ASID. The current ASID is an integer stored in a hidden register of the current CPU core, which cannot be accessed by software in the guest VM.

SEV allows the guest OS to decide, by setting or clearing the C-bit of the PTE, whether each virtual memory page is (treated as) private (encrypted with the guest's VEK) or shared (either encrypted with the host's VEK or unencrypted). For instance, when the C-bit of a page is set, memory reads from this virtual-physical mapping is considered encrypted with the guest VM's VEK, regardless of its true encryption status, and thus a memory read in that page will be decrypted using the VEK of the current ASID.

However, the hypervisor is able to manipulate the nested C-bit (nC-bit) in nPT. When the gC-bit (the C-bit of the gPT) conflicts with the nC-bit, AMD-SP encrypts the memory pages according to rules specified in Table 2.1: When gC-bit=0 and nC-bit=1, the page is encrypted with the hypervisor's VEK; when gC-bit=1, regardless of the nC-bit, the page is encrypted with the guest VM's VEK; when gC-bit=0 and nC-bit=0, the page is not encrypted. Following SME, the code pages are always considered private to the guest VM and thus is always encrypted regardless of the guest C-bits. Similarly, the gPT is also always encrypted with the guest's VEK, while the nPT is fully controlled by the hypervisor.

4.1.1.2 ASID-based TLB Isolation

ASID was originally introduced to avoid TLB flushes when the execution context switches between the guest VM and the hypervisor, which is achieved by extending each TLB tag with ASID. With the ASID capability, when observing activities like MOV-to-CR3, context switches, updates of `CR0.PG/CR4.PGE/CR4.PAE/CR4.PSE`, the hardware does not need to flush the entire TLB, but only the TLB entries tagged with the current ASID [6]. However, for the purpose of TLB isolation, the management of ASIDs for non-SEV VMs and SEV VMs is slightly different.

Non-SEV VMs. Each VCPU of a non-SEV VM may have different ASIDs, which can be assigned dynamically before each VMRUN. More specifically, before the hypervisor is about to resume a VCPU with VMRUN, it checks if the VCPU was the one running on this CPU core before the control was trapped into the hypervisor. If so, the hypervisor keeps the ASID of the VCPU unchanged and resumes the VCPU directly; if not, the hypervisor selects another ASID (from the ASID pool) and assign it to the VCPU. In the former case, TLB entries can be reused by the VCPU as its ASID is unchanged. However, in the latter

case, the residual TLB entries (tagged with ASID of the hypervisor or the previous VCPU) should not be reused.

SEV VMs. SEV processors rely on a similar strategy to isolate entries in the TLBs with ASID. However, instead of dynamically assigning an ASID to a VCPU before VMRUN, all VCPUs of the same SEV VM are assigned the same ASID at launch time, which should in theory remain the same during the entire life cycle of the SEV VM.

4.1.1.3 ASID-based Cache Isolation

On platforms that support SEV, cache lines are tagged with the VM's ASID indicating to which VM this data belongs, thus preventing the data from being misused by entities other than its owner [51]. When data is loaded into cache lines, according to the current ASID, AMD-SP automatically decrypts the data with the corresponding VEK and stores the ASID value into the cache tag. When a cache line is flushed or evicted, AMD-SP uses the ASID in the cache tag to determine which VEK to use when encrypting this cache line before writing it back to DRAM. The cache tag is also extended to include the C-bit [51]. Because the cache is now tagged with ASID and C-bit, cache coherence of the same physical address is not maintained if the two virtual memory pages do not have the same ASID and C-bit.

4.1.2 ASID Management

4.1.2.1 ASID Life Cycle

The hypervisor reserves a pool (*i.e.*, a range of integers) of available ASIDs for all VMs (we call all-ASID pool for simplicity), and a separate pool of ASIDs for SEV VMs (SEV-ASID pool). The maximum ID number of the all-ASID pool is determined by CPUID 0x8000000a[EBX] (*e.g.*, 32768, thus the available ASIDs are whole numbers between 1 and 32768). The maximum ID number of the SEV-ASID pool is determined by CPUID

0x8000001f[ECX] (*e.g.*, 15, which suggests the legal ASIDs for SEV VMs are 1 to 15). Note that ASID 0 is reserved for the host OS (i.e., hypervisor), and is also not allowed to be assigned to a VCPU for processors with or without SEV extensions [6].

On SEV platforms, the hypervisor uses `ACTIVATE` command to inform AMD-SP that a given guest is bound with an ASID and uses `DEACTIVATE` command to de-allocate an ASID from the guest. The hypervisor may re-allocate an existing ASID to another VM, if there is no available ASID in the SEV-ASID pool [7].

At runtime, when the processor runs under the guest mode, the guest VM's ASID is stored in the ASID register that is hidden from software; when the processor runs under the host mode, the register is set to 0, which is the hypervisor's ASID. The guest VM's ASID is stored at the VMCB during VMEXIT. After VMRUN the processor restores the ASID in the VMCB. The VMCB State Cache allows the processor to cache some guest register values between VMEXIT and VMRUN for performance enhancement. The physical address of the VMCB is used to perform access control of the VMCB State Cache. However, the VMCB clean field controlled by the hypervisor can be used to force the processor to discard selected cached values. For example, bit-2 of the VMCB clean field indicates that an ASID reload is needed; bit-4 of the clean field indicates fields related to nest paging are dirty and needed to be reloaded from the VMCB. Some VMCB fields are strictly not cached and the corresponding register values will be reloaded from the VMCB every time. For example, offset 058h of the VMCB is a TLB control field to indicate whether the hardware needs to flush TLB after VMRUN; this field is always uncached.

4.1.2.2 ASID Restrictions

Launch-time restrictions. On processors supporting SEV, the hypervisor cannot bind a current active ASID in the SEV-ASID pool to an SEV VM during launch time [7].

However, an adversary is able to deactivate the victim SEV VM and then activate an attacker SEV VM with the same ASID. The hardware requires the hypervisor to execute a `WBINVD` instruction and a `DF_FLUSH` instruction after deactivating an ASID and before re-activating it. The `WBINVD` flushes all modified cache lines and invalidates all cache lines. The `DF_FLUSH` instruction flushes data fabric write buffers of all CPU cores. If these instructions are not executed before associating the ASID with a new VM, a `WBINVD_REQUIRED` or `DFFLUSH_REQUIRED` error will be returned by the AMD-SP and the VM launch process will be terminated.

This restriction is critical to the isolation of cache lines. Otherwise, victim VM's residual cache data can be read by subsequent attacker VM. In particular, the attacker VM can use the `WBINVD` instruction to flush the cache data to memory. Cache lines belonging to victim VM will thus be encrypted with the attacker VM's VEK and then flushed into the memory. Subsequent reads to those memory data will return plaintext and thus allow the adversary to extract the data.

Run-time restrictions. After a VM is launched, the hypervisor can change its ASID during VMEXITs, by changing the ASID field of its VMCB, which will take effect when the VM is resumed. There is no additional hardware restriction at runtime. As such, it is possible to have two SEV VMs concurrently with the same ASID on the same machine, though the one with a wrong ASID will crash very soon.

Moreover, the VMCB also contains a field (090h) to indicate if the VM is an SEV VM or a non-SEV VM. Therefore, it is possible to first launch an SEV VM and a non-SEV VM with the same ASID, and then, during VMEXITs of the non-SEV VM, change the non-SEV VM into an SEV VM by setting the corresponding bit in the VMCB. We have experimentally confirmed this possibility on our testbed (as shown in Section 4.4.1). It suggests that the

hardware trusts the values of VMCB to determine (1) if the VM to be resumed is an SEV VM and (2) what ASID is associated with it. The hardware does not store this information to a secure memory region and use it for validation. The only additional validation performed by the AMD-SP is that the ASIDs of SEV VMs must fall into the valid ranges[3]. Therefore, while a VM was launched as a non-SEV VM, we can effectively (though momentarily) make it an SEV VM with the same ASID as another SEV VM.

4.1.2.3 "Security-by-Crash"

As the hypervisor has the liberty of changing the ASIDs of both SEV VMs and non-SEV VMs, security concerns arise when the hypervisor is not considered a trusted party. However, it is believed that when an SEV VM is resumed with an ASID different from its own, its subsequent execution will lead to unpredictable results and eventually crash the VM.

Specifically, to change the ASID of a VM (either an SEV and non-SEV VM), the hypervisor can directly edit the ASID field of the VMCB, set the VMCB clean-field to inform the hardware to bypass the VMCB State Cache, and then resume the VM with VMRUN. After the VM is resumed, if the `RFLAGS.IF` bit in the VMCB is set, the virtual address specified by the interrupt descriptor-table register (IDTR) will be accessed, because the guest OS will try to handle interrupts immediately; if the `RFLAGS.IF` bit is cleared, the instruction pointed to by NRIP—the next sequential instruction pointer—is going to be fetched and executed. However, in either case, the virtual address translation will cause problems.

First, any TLB entries remaining due to its previous execution becomes invalid because its ASID has been changed; the ASID tag in the TLB entries would not match. Second,

[3]Specifically, the valid ASID range of SEV VMs are divided so that the lower values are for SEV-ES VMs. CPUID Fn8000_001F[ECX] specifies valid SEV ASIDs and CPUID Fn8000_001F[EDX] specifies the minimum ASID values used for SEV (but SEV-ES-disabled) VMs.

a page table walk is unlikely to succeed, as its own page tables are encrypted using the VEK indexed by its own ASID. As a result, the top-level page table will be decrypted into meaningless bit strings. References to a "page table entry" of this page will trigger an exception to be handled by the guest OS. Finally, a handler of the guest OS is to be invoked to handle the exception. However, any reference of this handler will be decrypted using a wrong VEK, leading to a *triple fault* that eventually crashes the VM.

4.1.3 ASID Isolation Summary

We highlight a few key points of SEV's "security-by-crash" based memory isolation mechanisms.

- **ASID is used for access control.** ASID is the only identifier used for controlling accesses to virtual memory, caches, and TLBs. Once a VM is successfully resumed from VMEXIT, the CPU and AMD-SP only rely on the ASID (loaded from its VMCB) to validate memory requests.
- **ASID is managed by the hypervisor.** The hypervisor may assign any ASID (including the ASID of another active SEV VM) to an SEV or non-SEV VM during VMEXIT. The only restriction enforced by the hardware is that the ASID must fall into the range in accordance to the VM's SEV type.
- **Security is achieved by VM crash.** The security of the mechanism relies solely on the faults triggered during the execution of the VM if its ASID has been changed. The faults can be caused by memory decryption with an improper VEK during instruction fetches or page table walks.
- **Cache/TLB entries are flushed by the hypervisor.** The hypervisor controls whether and when to flush TLB and cache entries associated with a specific ASID. Only limited

constraints are enforced by the hardware during ASID activation. Misuse of these resources is possible.

4.2 CROSSLINE Attacks

The goal of our CROSSLINE attacks is to extract the memory content of the victim VM that is encrypted with the victim VM's VEK. We make no assumption of the adversary's knowledge of the victim VM, including its kernel version, the applications running in it, *etc.* The common steps of the CROSSLINE attacks are the following: (1) the adversary who controls the hypervisor launches a carefully crafted attacker VM; (2) the hypervisor alters the ASID of the attacker VM to be the same as that of the victim VM during VMEXITs; (3) the hypervisor prepares a desired execution environment for the attacker VM by altering its VMCB and/or its nPT; (4) the attacker VM resumes after VMRUN, allowing a *momentary execution* before it crashes. During the momentary execution, memory accesses from the attacker VM will trigger memory decryption using the victim VM's VEK.

Although the attacker VM crashes shortly—due to the ASID-based isolation in TLB, caches, and memory—we show that this momentary execution, though very brief, already enables the attacker VM to impersonate the victim VM and breach its confidentiality and integrity. Note that the only requirement of the victim VM at the time of the attack is that it has been launched and the targeted memory pages have been encrypted in the physical memory. Whether or not the victim VM is concurrently running during the attack is not important. Therefore, CROSSLINE is stealthy in that it does not interact with the victim VM at all. Detection of such attacks from the victim VM itself is unlikely.

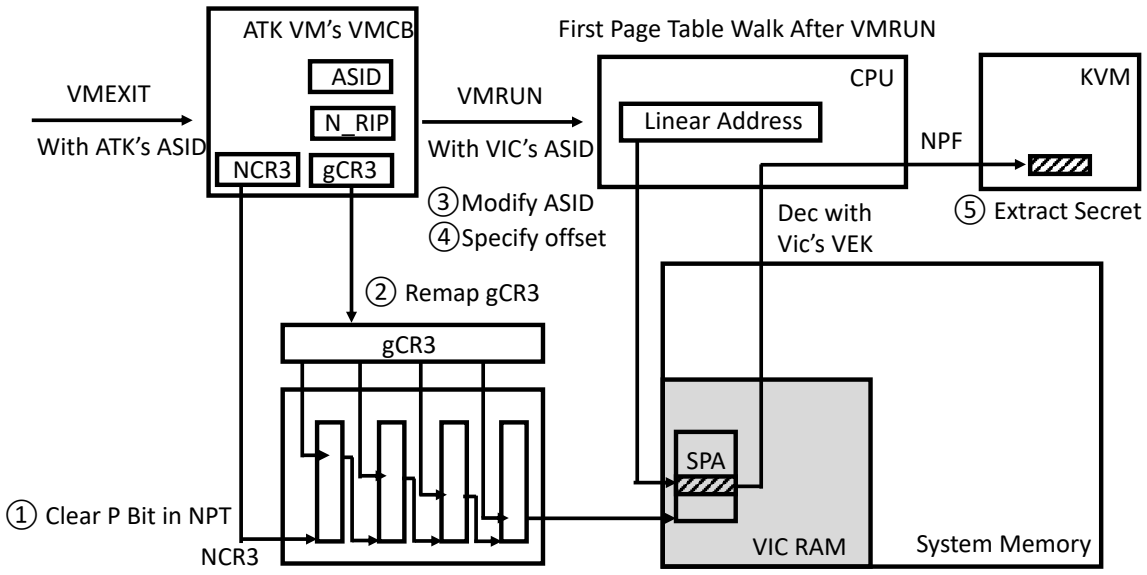


Figure 4.2: Workflow of CROSSLINE V1.

4.2.1 Variant 1: Extracting Encrypted Memory through Page Table Walks

The CROSSLINE V1 explores the use of nested page table walks during the momentary execution to decrypt the victim VM's memory protected by SEV. To ease the description, let the victim VM's ASID be 1 and the attacker VM's ASID be 2. We use SPFN_0 to denote the system page frame number of the targeted memory page encrypted with the victim VM's VEK. We use sPA_0 to denote the system physical address of one 8-byte aligned memory block on SPFN_0, which is the target memory the adversary aims to read. The workflow of CROSSLINE V1 is shown in Figure 4.2. When the hypervisor handles a VMEXIT of the attacker VM, the following steps are executed:

① **Clear the Present bits.** The hypervisor alters the attacker VM's nPT to clear the Present bits of the PTEs of all memory pages. Thereafter, any memory access from the attacker VM

after VMRUN will trigger a nested page fault, because the mapping from gPA to sPA in the NPT is missing.

② **Remap the current gCR3 of the attacker VM.** The hypervisor remaps the gCR3 of the current process in the attacker VM by altering the nPT. Now the gCR3 maps to SPFN_0. The hypervisor then sets the Present bit of this new mapping in the nPT.

③ **Modify the attacker VM's VMCB.** The hypervisor changes the attacker VM's ASID field in the VMCB to the victim VM's ASID (from 2 to 1 in this example).

④ **Specify the targeted page offset.** Before resuming the attacker VM with VMRUN, the hypervisor also modifies the value of NRIP in VMCB to specify which offset (*i.e.*, sPA_0) of the target page (*i.e.*, SPFN_0) to decrypt. Specifically, in a 64-bit Linux OS, bits 47 to 12 of a virtual address are used to index the page tables: bits 47-39 for the top-level page table; bits 38-30 for the second-level; bits 29-21 for the third; and bits 20-12 for the last-level page table. Each 4KB page in the page table has 512 entries (8 bytes each) and each entry contains the page frame number of the memory page of next-level page table or, in the case of the last-level page table, the page frame number of the target address. CROSSLINE V1 exploits the top-level page table walk to decrypt one 8-byte block each time. To control the offset of the 8-byte block within the page, the adversary modifies the value of NRIP stored in the VMCB so that its bit 47-39 can be used to index the top-level page table. The algorithm to choose NRIP properly is specified in Algorithm 1. Specifically, if the offset is less than 0x800, the NRIP is set to be in the range of 0x0000000000000000 - 0x00007fffffffffff (canonical virtual addresses of user space); if the offset is greater than or equal to 0x800, the NRIP is set to be in the range of 0xffff800000000000 - 0xffffffffffffffff (canonical virtual addresses of kernel space).

Algorithm 1 Determine NRIP when dumping one layer of page table (4096 bytes)

```
initialization  while dumping the page do
    try to dump 8-byte memory block sPA_0   if sPA_0 % 0x1000 < 0x800 then
        NRIP = 0x8000000000* (sPA_0 % 0x1000 / 0x8)
    else
        NRIP = 0xffff000000000000 + 0x8000000000* (sPA_0 % 0x1000 / 0x8)
    end
end
```

⑤ **Extract secrets from nested page faults.** After VMRUN, the resumed attacker VM immediately fetches the next instruction to be executed from the memory. This memory access is performed with ASID=1 (*i.e.*, the victim VM's ASID). The address translation is also performed with the same ASID. As the TLB does not hold valid entries for address translation, and thus an address translation starts with a page table walk from the gCR3, which maps to SPFN_0 in the nPT. Therefore, an 8-byte memory block on SPFN_0, whose offset is determined by bit 47-39 of the virtual address of the instruction, is loaded by the processor as if it is an entry of the page table directory. As long as the corresponding memory block follows the format of a PTE (to be described shortly), the data can be extracted and notified to the adversary as the faulting address (encoded in the EXITINFO2 field of VMCB).

4.2.1.1 Dumping Victim Page Tables

A direct security consequence of CROSSLINE V1 is to dump the victim VM's entire guest page table, which is deemed confidential as page-table pages are always encrypted in SEV VMs regardless of the C-bit in the PTEs.

To dump the page table, the adversary first locates the root of the victim VM's guest page table specified by its gCR3. She can do so by monitoring the victim VM's page access sequence using page-fault side channels. Specifically, during the victim VM's VMEXIT,

the adversary clears the Present bit of all page entries of the nPT of the victim VM, evicts all the TLB entries, invalidates the nPT entries cached by nTLB and PWC. After VMRUN, the victim VM immediately performs a page table walk. The gPA of the first page to be accessed is stored in its gCR3. The adversary thus learns the gPA of the root of the guest page table. Once each of the entries of the root page table is extracted by CROSSLINE V1, the rest of the page table can be decrypted one level after another.

Evaluation. We evaluated this page table dump attack using CROSSLINE on a blade server with an 8-Core AMD EPYC 7251 Processor. The host OS runs Ubuntu 64-bit 18.04 with Linux kernel v4.20 and the guest VMs run Ubuntu 64-bit 18.04 with Linux kernel v4.15 (SEV supported since v4.15). The QEMU version used was QEMU 2.12. The victim VMs were SEV-enabled VMs with 4 virtual CPUs, 4 GB DRAM and 30 GB disk storage. The attacker VMs were SEV-enabled VMs with only one virtual CPU, 2 GB DRAM and 30 GB disk storage. All the victim VMs were created by the ubuntu-18.04-desktop-amd64.iso image with no additional modification. The guest OS version does not matter in our attack.

To decrypt one 8-byte memory block, the adversary needs to launch the attacker VM, let it run until a VMEXIT, change its ASID, clear the Present bit of all PTEs. Roughly, it takes 2 seconds to decrypt one 8-byte memory block (which includes time to deactivate the ASID, reboot the VM, and clear the Present bit of all PTEs).

To speed up the memory decryption, the adversary could perform the following *VMCB rewinding attack*. Particularly, after extracting one 8-byte block through a VMEXIT caused by the nested page fault, the adversary could continue to decrypt the next 8-byte block without rebooting the attacker VM. To do so, the adversary directly repeats the attack steps by rewinding the VMCB of the attacker VM to the previous state and changing the NRIP to perform the next round of attack. With this approach, we found the average time to decrypt a

4KB memory page (with 1 attacker VM in 500 trials) was only 39.580ms (with one standard deviation of 4.26ms).

4.2.1.2 Reading Arbitrary Memory Content

Beyond page tables, the adversary could also extract regular memory pages of the victim VM. For example, if the data of an 8-bytes memory block is `0x00 0x00 0xf1 0x23 0x45 0x67 0x8e 0x7f`, the extracted data through page fault is `0x712345678`; if the data is `0x00 0x00 0x0a 0xbc 0xde 0xf1 0x20 0x01`, the extracted data is `0xabcdef12`. However, as CROSSLINE V1 only reveals the encrypted data as a page frame number embedded in the PTE, such memory decryption only works on 8-byte aligned memory blocks (*i.e.*, the begin address of the block is a multiple of 8 and the size of the block is also 8 bytes) that conforms to the format of a PTE.

Concretely, as shown in Figure 4.3, the 8-byte memory block to be extracted from CROSSLINE, must satisfy the following requirements: The Present bit (bit 0) must be 1; Bits 48-62 must be all 0s, and Bits 7-8 are both 0s (optional). This is because the Present bit must be 1 to trigger nested page fault. Otherwise, non-present faults in the guest VM will be handled without involving the hypervisor. Bits 48-62 are reserved and must be 0. The Page Size (PS) bit (bit-7) is used to determine the page size (*e.g.*, 4KB vs. 2MB); the Global Page (G) bit (bit-8) is used to indicate whether the corresponding page is a global page. These 2 bits can only be set 1 in the last level of the page table. Therefore, if CROSSLINE V1 generates page faults at the top-level page table, they must be set 0. However, we find it possible to configure the nPT so that the first three levels of the guest page table walk all pass successfully, and only trigger the nested page fault at the last-level page table. In this way, the target memory block can be regarded as a PTE of the last-level page table and hence these two bits are not restricted to be 0s.

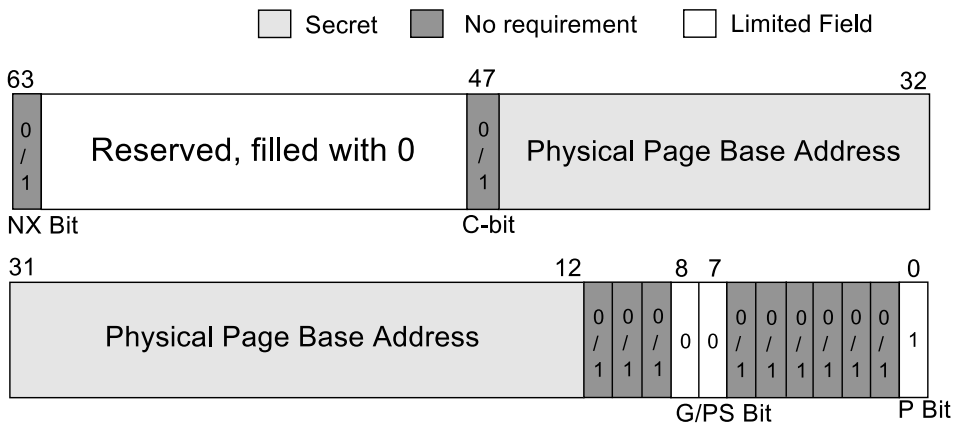


Figure 4.3: Valid PTE format.

Performance evaluation. The speed of memory decryption for arbitrary memory content is the same as dumping page tables, as long as the they are of PTE format. If the target block does not conform to the PTE format, a triple fault takes place instead of nested page fault, in which case the adversary could perform the VMCB rewinding attack and target another memory block in the next round of attacks.

Percentage of readable memory blocks. We studied the binary file of ten common applications, `python` 2.7, `OpenSSH` 7.6p1, `perl` 5.26.1, `VIM` 8.0.1453, `tcpdump` 4.9.3, `patch` 2.7.6, `grub-install` 2.02.2, `sensors` 3.4.0 , `Nginx` 1.14.0, and `diff` 3.6, which are installed from the default package archives in Ubuntu 18.04 (64-bit). The percentages of 8-byte aligned memory blocks that can be directly read using this method is 1.00%, 1.53%, 1.79%, 1.81%, 2.10%, 3.50%, 4.00%, 5.88%, 6.10%, and 6.50%. While they only account for a small portion of the whole memory space, they leak enough information for process fingerprinting purposes.

4.2.2 Variant 2: Executing Victim VM's Encrypted Instructions

In CROSSLINE V2, we show that, when certain conditions are met, it is possible for the attacker VM to momentarily execute a few instructions that are encrypted in the victim VM's memory. Apparently, CROSSLINE V2 is more powerful than the previous variant. Fortunately, the only prerequisite of CROSSLINE V2 is the consequence of CROSSLINE V1.

Similar to the settings in the previous attack variant, two SEV VMs were configured so that the ASID of the victim VM is 1 and the ASID of the attacker VM is 2. We assume that the attacker VM aims to execute one instruction—"`movl $2020, %r15d`"—in the victim VM's encrypted memory. Let the virtual address of this target instruction be gVA_0 and the corresponding gCR3 of the target process be $gCR3_0$. The adversary's strategy is to follow the common steps of CROSSLINE attacks and manipulate the nPT of the attacker VM so that it finishes a few nested page table walks to successfully execute this instruction. More specifically, CROSSLINE V2 can be performed in the following steps:

① **Prepare nPT.** The hypervisor clears the Present bit of all PTEs of the attacker VM's nPT. It also prepares valid mappings for the gVA_0 to the physical memory encrypted with the victim's VEK. To do so, the hypervisor needs to prepare five gPA to sPA mappings (for the gPFNs of the four levels of the gPT and the instruction page), respectively.

② **Set NRIP.** The hypervisor sets NRIP as gVA_0. It also clears the Interrupt Flag of the RFLAGs register (RFLAGS.IF) in the VMCB, so that the attacker VM directly executes the next instruction specified by NRIP, instead of referring to Interrupt-Descriptor-Table Register.

③ **Change ASID.** The hypervisor changes the attacker VM's ASID to the victim's ASID, marks the VMCB as dirty, and resumes the attacker VM with VMRUN. During the next VMEXIT, the value of $\%r15$ has been changed to \$2020, which means the attacker VM has successfully executed an instruction that is encrypted with the victim's VEK.

These experiments suggest that CROSSLINE allows the attacker VM to execute some instruction of the victim VM. We exploit this capability to construct decryption oracles and encryption oracles.

4.2.2.1 Constructing Decryption Oracles

A decryption oracle allows the adversary to decrypt an arbitrary memory block encrypted with the victim's VEK. With CROSSLINE V2, the attacker VM executes one instruction of the victim VM to decrypt the target memory.

The first step of constructing a decryption oracle is to locate an instruction in the victim VM with the format of "`mov (%`reg_1`),%`reg_2", which loads an 8-byte memory block whose virtual address is specified in reg_1 to register reg_2. As most memory load instructions follow this format, the availability of such an instruction is not an issue. The adversary can leverage CROSSLINE V1 to scan the physical memory of the victim VM, in the hope that the readable memory blocks contain such a 3-byte instruction. Alternatively, if the kernel version of the victim VM is known, the adversary can scan the binary file of the kernel image to locate this instruction and then obtain its runtime location by reading the gPT, which can be completely extracted by CROSSLINE V1.

Let the virtual address of this instruction be gVA_0, its corresponding system physical address be sPA_0, and the gCR3 value of the process in the victim VM be gCR3_0. The virtual address and the system physical address of the target memory address to be decrypted are gVA_1 and sPA_1. Note since the adversary is able to extract the gPT of the victim, the

corresponding translation for gVA_0 and gVA_1 can be obtained. Then following the three steps outlined above, during a VMEXIT of the attacker VM, the adversary prepares the nPT of the attacker VM (including one mapping for $gCR3_0$, four mappings for gVA_0, and four mappings for gVA_1), configures the VMCB (including NRIP, ASID, the value of $\%reg_1$), and then resumes the attacker VM.

In the next VMEXIT, the adversary is able to extract the secret stored in sPA_1 by checking the value of $\%reg_2$. The adversary can immediately perform the next round of memory decryption. The system physical page frame number can be manipulated in the last-level nPT and the page offset can be controlled in $\%reg_1$.

Performance evaluation. We measured the performance of the decryption oracle described above for decrypting a 4KB memory page. With only one attacker VM, the average decryption time (of 5 trials) for a 4KB page was 113.6ms with one standard deviation of 4.3ms. Note the decryption speed is slower than the optimized version of CROSSLINE V1, but the decryption oracle constructed with CROSSLINE V2 is more powerful as it is not limited by the format of the target memory block.

4.2.2.2 Constructing Encryption Oracles

An encryption oracle allows the adversary to alter the content of an arbitrary memory block encrypted with the victim's VEK to the value specified by the adversary. With CROSSLINE V2, an encryption oracle can be created in ways similar to the decryption oracle. The primary difference is that the target instruction is of the format "`mov` $\%reg_1$`,(`$\%reg_2$`)`", which moves an 8-byte value stored in reg_1 to the memory location specified by reg_2.

With an encryption oracle, the adversary could breach the integrity of the victim VM and force the victim VM to (1) execute arbitrary instruction, or (2) alter sensitive data, or

(3) change control flows. Note that our encryption oracle differs from those in the prior works [23, 29, 61] as it does not rely on SEV's memory integrity flaws.

Performance evaluation. We measured the performance of the encryption oracle by the time it takes to updates the content of a 4KB memory page. The average time of 5 trials was 104.8ms with one standard deviation of 6.1ms. Note in a real-world attack, the attacker may only need to change a few bytes to compromise the victim VM, which means the attack can be done within 1ms.

4.2.2.3 Locating Decryption/Encryption Instructions

In the previous experiments, we have already shown that once the instructions to perform decryption and encryption can be located, the construction of decryption and encryption oracles is effective and efficient. Next, we show how to locate such decryption/encryption instructions to bridge the gap towards an end-to-end attack.

Specifically, on the victim VM, an OpenSSH server (SSH-2.0-OpenSSH-7.6p1 Ubuntu-4ubuntu0.1) is pre-installed. *First*, the adversary learns the version of the OpenSSH binary by monitoring the SSH handshake protocol. More specifically, the adversary who controls the hypervisor and host OS monitors the incoming network packets to the victim VM to identify the SSH `client_hello` message. The victim VM would immediately respond with an SSH `server_hello` message, which contains the version information of the OpenSSH server. As these messages are not encrypted, the adversary could leverage this information to search encryption/decryption instructions offline from a local copy of the binary.

Second, the adversary extracts the gCR3 of the *sshd* process. To do so, upon observing the `server_hello` message, the adversary immediately clears the Present bits of all PTEs of the victim VM. The next memory access from the *sshd* process will trigger an NPF

VMEXIT, which reveals the value of gCR3. We empirically validated that this approach allows the adversary to correctly capture *sshd*'s gCR3, by repeating the above steps 50 times and observing correct gCR3 extraction every time.

Third, the adversary uses CROSSLINE V1 to dump a portion of the page tables of *sshd* process. More specifically, the adversary first dumps the 4KB top-level page-table page pointed to by gCR3; she identifies the smallest offset of this page that represents a valid PTE, and then follow this PTE to dump the second-level page-table page. The adversary repeats this step to dump all four levels of page tables for the lowest range of the virtual address. In this way, the adversary could obtain the physical address corresponding to the base virtual address of the OpenSSH binary.

Fourth, with the knowledge of the memory layout of the code section of the OpenSSH binary, the adversary can calculate the physical address of the decryption/encryption instructions within the OpenSSH binary. In our demonstrated attack, the adversary targets two instructions inside the `error` function of OpenSSH, "`mov (%rbx),%rax`" for decryption and "`mov %rax,(%r12)`" for encryption. The offsets of the two instructions are `0xca9a` and `0xca18`, respectively.

Performance evaluation. We measured the time needed to locate these two instructions. Once the adversary has intercepted the SSH handshake messages, it takes on average 504.74ms (over 5 trials) to locate these two instructions.

4.2.3 Discussion on Stealthiness and Robustness

CROSSLINE attacks are stealthy. The attacker VM and the victim VM are two separate VMs. They have different NPTs and VMCBs. Therefore, any state changes made in the attacker VM are not observable by the victim. As such, it is impossible for victim

VM to sense the presence of the attacker VM. In contrast to all known attacks to SEV, CROSSLINE cannot be detected by running a detector in the victim VM. More interestingly, the adversary can rewind the attacker VM's VMCB to eliminate the side effects caused by the attacker VM's attack behaviors (*e.g.*, triggering a NPF with non-PTE format or executing an illegal instruction). This method also increases the robustness of the attack: Even if the encryption/decryption instructions are not correctly located, CROSSLINE V2 will not affect the execution of the victim VM. Therefore, the adversary can perform the attack multiple times until succeeds.

4.3 Applicability to SEV-ES

4.3.1 Overview of SEV-ES

To protect VMCB during VMEXIT, SEV-ES was later introduced by AMD [49]. With SEV-ES, a portion of the VMCB is encrypted with authentication. Therefore, the hypervisor can no longer read or modify arbitrary register values during VMEXITs. To exchange data between the guest VM and the hypervisor, a new structure called Guest Hypervisor Control Block (GHCB) is shared between the two. The guest VM is allowed to indicate what information to be shared through GHCB.

VMEXITs under SEV-ES modes are categorized into Automatic Exits (AE) and Non-Automatic Exits (NAE). AE VMEXITs (*e.g.*, those triggered by most nested page faults, by the `PAUSE` instruction, or by physical and virtual interrupts) are VMEXITs, which do not need to expose register values to the hypervisor. Therefore, AE VMEXITs directly trigger a VMEXIT to trap into the hypervisor. To enhance security, NAEs (*e.g.*, those triggered by `CPUID`, `RDTSC`, `MSR_PROT` instructions) are first emulated by the guest VM instead of the hypervisor. Specifically, NAEs first trigger #VC exceptions, which are handled by the

guest OS to determine which register values need to be copied into the GHCB. This NAE VMEXIT will then be handled by hypervisor that extracts the register values from the GHCB. After the hypervisor resuming the guest in VMRUN, the #VC handler inside the guest OS reads the results from the GHCB and copies the relevant register states to corresponding registers.

SEV-ES VMs can run concurrently with SEV VMs and non-SEV VMs. After VMEXIT, the hardware recognizes an SEV-ES VM by the SEV control bits (bit-2 and bit-1 of 090h) in the VMCB [6]. Therefore, the hypervisor may change the SEV type (from an SEV VM to an SEV-ES VM) during VMEXIT. The legal ASID ranges of SEV-ES and SEV VMs, however, are disjoint, and thus it is not possible to run an SEV-ES VM with an ASID in the range of SEV VMs.

4.3.1.1 VMCB's Integrity Protection

With SEV-ES, the VMCB is divided into two separate sections, namely the control area and the state save area (VMSA) [6]. The control area is unencrypted and controlled by the hypervisor, which contains the bits to be intercepted by the hypervisor, the guest ASID (058h), control bits of SEV and SEV-ES (090h), TLB control (058h), VMCB clean bits (0C0h), NRIP (0C8h), the gPA of GHCB (0A0h), the nCR3 (0B0h), VMCB save state pointer (108h), *etc.* The state save area is encrypted and integrity protected, which contains the saved register values of the guest VM. The VMCB save state pointer stores the system physical address of VMSA, the encrypted memory page storing the state save area.

The integrity-check value of the state save area is stored in the protected DRAM, which cannot be accessed by any software, including the hypervisor [6]. At VMRUN, the processor performs an integrity check of the VMSA. If the integrity check fails, VMRUN terminates with errors [6]. Because the integrity-check value (or the physical address storing the value)

is not specified by the hypervisor at VMRUN, we conjecture the value is index by the system physical address of the VMSA. Therefore, a parked virtual CPU is uniquely identified by the VMSA physical address.

4.3.2 CROSSLINE V1 on SEV-ES

The primary challenge to apply CROSSLINE on SEV-ES machines is to bypass the VMSA check. Directly resuming the attacker VM using the victim's ASID would cause VMRUN to fail immediately, because the VMSA integrity check takes place before fetching any instructions in the attacker VM. Since the attacker VM's VMSA is encrypted using the VEK of the attacker VM, when resuming the attacker VM with the victim's ASID, the decryption of VMSA leads to garbage data, crashing the attacker VM immediately.

Therefore, to perform CROSSLINE V1, the adversary must change the save state pointer (0108h) of the attacker VM's VMCB so that the attacker VM will reuse the victim VM's VMSA. As such, the attacker VM cannot change the register values that are stored in the VMSA, which includes RIP, gCR3, and all general-purpose registers (if not exposed in the GHCB). Therefore, with SEV-ES, the adversary is no longer able to arbitrarily control the execution of the attacker VM by simply manipulating its NRIP in its VMCB's control area [6].

However, by pausing victim's VCPU during VMEXIT and changing attacker's VMSA pointer (0108h) to victim's VMSA, the adversary is still able to perform CROSSLINE V1 on SEV-ES VMs to achieve the same goal—extracting the entire gPT or decrypting any 8-byte memory block conforming to a PTE format. To show this, we have performed the following experiments:

Two SEV-ES VMs were launched. The ASID of the victim VM is set to be 1 and that of the attacker VM is 2. The hypervisor pauses the victim VM at one of its VMEXIT, so that its VMSA is not used by itself. The attack is performed in the following steps:

① **Prepare nPT.** During the VMEXIT of the attacker VM, the hypervisor clears the Present bits of the all PTEs in the attacker VM's nPT.

② **Manipulate the attacker VM's VMCB.** The hypervisor first changes the attacker VM's ASID from 2 to 1. It also informs the hardware to flush all TLB entries of the current CPU, by setting the TLB clearing field (058h) in the VMCB control area. Finally, it changes the VMCB save area pointer to point to the victim's VMSA.

③ **Resume the attacker VM.** Because the attacker VM runs with the victim's ASID, the victim's VMSA is decrypted correctly. The integrity check also passes, as no change is made in the VMSA, including its system physical address. Once resumed, the attacker VM will try to fetch the first instruction determined by RIP (in VMSA) or the IDTR using the victim's VEK. Since there is no valid TLB entry, the processor has to perform a guest page table walk to translate the virtual address to the system physical address. A nested page fault can be observed with the faulting address being the victim VM's gCR3 value.

④ **Remap gCR3 in nPT.** When handling this NPF VMEXIT, the hypervisor remaps the gCR3 in the nPT to the victim VM's memory page to be decrypted. The Present bits of the corresponding nested PTEs are set to avoid another NPF of this translation. Moreover, the `EXITINTINFO` field in the unencrypted VMCB control area needs to be cleared to make sure the attacker VM complete the page table walk. After resuming the attacker VM, an NPF for the translation of another gPA (embedded in the target memory block) will occur, which reveals the content of the 8-byte aligned memory block if it follows the format of a PTE.

⑤ **Reuse the VMSA.** The hypervisor repeats step ④ so that its gCR3 is remapped to the next page to be decrypted in the victim VM. Then, the next NPF VMEXIT reveals the corresponding memory block. This could work because the attacker VM has not successfully fetched a single instruction yet; it is trapped in the first page table walk (more specifically, the first nested page table walk of the first gPA). Therefore, the VMSA is not updated and no valid TLB entry is created. During the remapping of gCR3, the hypervisor is able to invalidate the previously generated entry in the nTLB. Thus, from the perspective of the attacker VM, step ④ does not change its state. Therefore, the attacks can be carried out repeatedly.

⑥ **Handling triple faults.** In step ④ or ⑤, if the targeted 8-byte memory block does not conform to the PTE format, a triple fault VMEXIT (error code 0x7f) will be triggered instead of the NPF VMEXIT. The adversary can continue to decrypt the next page if this happens. However, after a triple fault, the RIP in the VMSA has been updated to the fault handler to deal with the fault. As such, resuming from a triple fault will lead to the decryption of a different offset of the target page. However, the attack can still continue.

Resuming the victim VM. After performing CROSSLINE V1, the VMSA of the victim VM is still usable by the victim. We empirically validated this by resuming the victim VM after the attacker VM has used this VMSA to decrypt several memory blocks and has encountered both nested page faults and triple faults. The victim VM was resumed successfully, without observing any faults or abnormal kernel logs.

To better understand the victim VM's state changes when its VMSA is used by the attacker VM, we instrumented the hypervisor to check which regions of the encrypted VMSA have been changed after the attacker VM has performed several rounds of CROSSLINE V1, which triggers both nested page faults and triple faults. The result shows that the entire VMSA remains the same, except the value of CR2, which stores the most recent faulting

address. The change of the CR2 value does not affect the execution of the victim VM as this value is not used by the guest OS after NPFs.

Controlling page offsets. Because the integrity protection of VMSA prevents the adversary from controlling the RIP after VMRUN, the page offset of the memory blocks to be decrypted cannot be controlled in CROSSLINE V1. However, the adversary may resume the victim VM and allow it to run till a different RIP is encountered. In total, 512 different RIPs are needed to decrypt any memory blocks conforming to the PTE formats. To diversify the exploited RIPs, one strategy is to pause the victim when the VMEXIT is a NPF-triggered AE. When VMEXITs are NAEs or interrupt-triggered AEs, the next instruction to be executed after VMRUN is an instruction of the #VC handler, whose virtual address is fixed in the kernel address space. To differentiate NPF-triggered AEs and interrupt-triggered AEs, although the adversary cannot read the RFLAG.IF directly, which indicates pending interrupts, she can inspect Bit 8 (V_IRQ) of the Virtual Interrupt Control field (offset 60h) in the unencrypted VMCB control area. Moreover, as two consecutive NPF-triggered AEs may be caused by the same RIP, it is preferred to pause the victim VM after a few AEs. To trigger more NPF VMEXITs, one could periodically unset the Present bit of all PTEs of the victim VM.

With these strategies in place, we empirically evaluated the time needed for the adversary to find all 512 offsets. In our test, we let the victim VM run a build-in program of Ubuntu Linux, called “cryptsetup benchmark". The attack can be performed on any level of the page tables; bits 47-39, 38-30, 29-21, and 20-12 of the same RIP can all be used as the page offset by the attacker. Therefore, with any RIP, there are $1\sim4$ different offsets that the attacker may use to extract data on any encrypted page. The experiments were performed in the following manner: Each round of the experiments, the cryptsetup benchmark were run

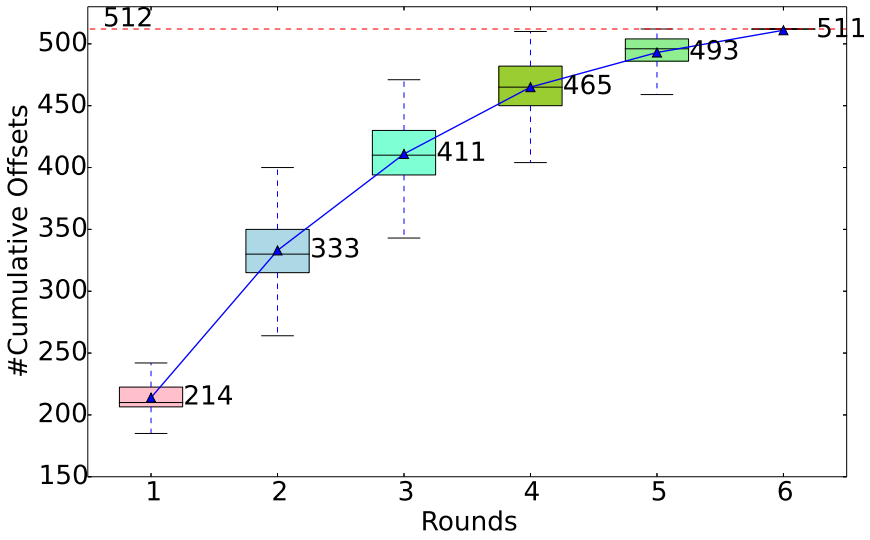


Figure 4.4: Covered offsets after N rounds.

several times and each time with a different address space layout due to ASLR; every 30 seconds, the adversary unset all Present bits of the victim VM to trigger NPFs; the adversary pauses the victim VM every 13 AE VMEXITs to extract one RIP. The adversary concludes the round of monitoring after 60 seconds. In total, 15 rounds of experiments were conducted. Figure 4.4 shows the number of offsets that can be covered after N rounds of experiments, where $N = 1$ to 6. Each data point is calculated over all combinations of selecting N rounds from the 15 rounds, *i.e.*, $C(15,N)$, of data collected in the experiments above. Specifically, on average, after 5 rounds of experiments, the adversary could obtain 493 offsets; after 6 rounds, she could obtain 511 offsets (out of the 512 offsets). These experiments show that when the victims run an application that has diverse RIPs (*i.e.*, not running in idle loops), the adversary has a good chance of performing CROSSLINE V1 on almost all page offsets after some efforts (in these experiments, after 6 minutes of the victim's execution).

Performance evaluation. We have evaluated the attack mentioned above on a workstation with an 8-Core AMD EPYC 7251 Processor. The motherboard of our testbed machine was GIGABYTE MZ31-AR0, with which we successfully configured Fn8000_001F[EDX] to return 5, which means ASID 1 to 4 were reserved for SEV-ES VMs. Since the source code supporting SEV-ES for both host OS and guest OS has not been added into the mainstream Linux kernel yet, we used the source code provided in the SEV-ES branch of AMD's official repositories for SEV, which is available on Github [9]. The kernel version for the host and guest were branch sev-es-5.1-v9. The QEMU version used was QEMU sev-es-v4 and the OVMF version was sev-es-v11. Both victim VMs and attacker VMs were configured as SEV-ES-enabled VMs with 1 virtual CPU, 2 GB DRAM and 30 GB disk storage. All VMs were created by the kernel image generated from sev-es-5.1-v9 branch without any additional modification.

On average over 200 trials, it takes 2.0ms to decrypt one 8-byte memory block, which is slower than the attack against SEV VMs (0.077ms per block). This is because the AMD-SP must calculate the hash of the VMSA and store it to the secure memory region during VMEXITs, and validate its integrity after each VMRUN. This happens in between of decrypting two memory blocks.

4.3.3 CROSSLINE V2 on SEV-ES

Applying CROSSLINE V2 on SEV-ES would be challenging, because with the encrypted VMCB, RIP is no longer controlled by the adversary. As such, the attacker VM will resume from the RIP stored in the VMSA, which prevents the attacker VM from executing arbitrary instructions. Moreover, constructing useful encryption or decryption oracles requires the manipulation of specific register values, which is only possible without SEV-ES.

Table 4.1: Demonstrated attacks against SEV. I/O Interaction: the attack requires interaction with applications inside the victim VM through I/O operations (*e.g.*, Network, disk). Stealthiness: the attack cannot be detected by the victim VM.

Research Papers	I/O Interaction	Breach Confidentiality	Breach Integrity	Stealthiness	Mitigated by
Du *et al.* [29]	✓	✗	✓	✗	SEV-SNP
Buhren *et al.* [23]	✓	✓	✗	✗	SEV-SNP
Wilke *et al.* [93]	✓	✓	✓	✗	SEV-SNP
Werner *et al.* [91]	✓	✓	✗	✗	SEV-ES
Hetzelt & Buhren [39]	✓	✓	✓	✗	SEV-SNP
Morbitzer *et al.* [69]	✓	✓	✗	✗	SEV-SNP
Morbitzer *et al.* [68]	✓	✓	✗	✗	SEV-SNP
Li *et al.* [61]	✓	✓	✓	✗	SEV-SNP
CROSSLINE V1	✗	✓	✗	✓	SEV-SNP
CROSSLINE V2	✗	✓	✓	✓	SEV-ES

4.3.4 Discussion on Stealthiness

Unlike CROSSLINE on SEV, to attack SEV-ES machines, the attacker VM must reuse the victim VM's VMSA. However, CROSSLINE V1 is still stealthy and undetectable by the victim VM for two reasons. First, the attack only alters the CR2 field of the victim's VMSA. As this field is not examined by the guest OS after resumption from a NPF, the victim VM cannot detect the anomaly. Second, even if the guest OS is modified, the change of the CR2 cannot be detected, because the AE NPFs are directly trapped into the hypervisor, such that the guest OS does not have a chance to record the original value of CR2 to be compared with.

We summarize the attacks against SEV, their exploited vulnerabilities, the attack consequences, and the stealthiness of the attacks in Table 7.2.

4.4 Discussion

4.4.1 A New Variant: Reusing Victim's TLB Entries

First, we discuss a proof-of-concept attack that extends the other two variants. In particular, in this attack, we show that the ASID-based TLB isolation can be breached. There were two VMs involved: the victim VM is an SEV VM whose ASID is 1; the attacker VM is a non-SEV VM whose ASID is 16. Both VMs only have one VCPU, which are configured by the hypervisor to run on the same logical CPU core. We assume the victim VM executes the following code snippet:

```
d83:41 bb e4 07 00 00    mov     $0x7e4,%r11d
d89:41 bc e4 07 00 00    mov     $0x7e4,%r12d
d8f:0f a2                cpuid
d91:eb f0                jmp     d83
```

Specifically, the code updates the values of `%r11d` and `%r12d`, and then executes a `CPUID` to trigger a VMEXIT. Following the common steps of CROSSLINE, the adversary launches an attacker VM, changes its ASID during VMEXIT, sets the NRIP of the attacker VM to the virtual address of the code snippet above, changes offset 090h of VMCB to make it an SEV VM, and resumes the attacker VM. Unlike CROSSLINE V1 and CROSSLINE V2, the nPT of the attacker VM is not changed in this step. Therefore, if the attacker VM performs a page table walk, a NPF will be triggered.

Interestingly, the execution of the attacker VM triggers `CPUID` VMEXITs before a triple fault VMEXIT crashes it. Since no NPF is observed, the attacker VM apparently does not perform any page table walk. However, during the attacker VM's `CPUID` VMEXITs, we observe that the values of `%r11d` and `%r12d` have been successfully changed to `$0x7e4`. It is clear that the two MOV instructions and the subsequent `CPUID` instruction have been executed by the attacker VM. This is because the attacker VM was able to use the victim

VM's TLB entries left in the TLB to translate the virtual address of the instructions. The triple fault might be caused by code executed outside the page, whose translation is not cached in the TLB.

While the consequences of this attack are close to V2, it highlights the following flaws in AMD's TLB isolation between guest VMs: (1) ASIDs serve as the only identifier for access controls to TLBs, which can be *forged* by the hypervisor, and (2) TLBs cleansing during VM context switch is performed at the discretion of the hypervisor, which may be *omitted* intentionally. We leave the exploration of this TLB problem to our proposed work.

4.4.2 Applicability to SEV-SNP

To address the attacks against SEV that exploit memory integrity flaws, AMD recently announced SEV-SNP [50] and released a whitepaper describing its high-level functionality in January, 2020 [8]. The key idea of SEV-SNP is to provide memory integrity protection using a Reverse Map Table (RMP). An RMP is a table indexed by system page frame numbers. One RMP is maintained for the entire system. Each system page frame has one entry in the RMP, which stores information of the page state (*e.g.*, hypervisor, guest-invalid, guest-valid) and ownership (*i.e.*, the VM's ASID and the corresponding gPA) of the physical page. The ownership of a physical page is established through a new instruction, `PVALIDATE`, which can only be executed by the guest VM. Therefore, the guest VM can guarantee that each guest physical page is only mapped to one system physical page; by construction, RMP allows each system physical page to have only one validated owner.

After each nested page table walks that leads to a system physical page belonging to an SEV-SNP VM (and also some other cases), an RMP check is to be performed. The RMP check compares the owner of the page (*i.e.*, the ASID) with the current ASID and compares

the recorded gPA in the RMP entry with the gPA of the current nPT walk. If a mismatch is detected, a nested page fault will be triggered.

- **CROSSLINE V1 on SEV-SNP.** When applying CROSSLINE V1 on SEV-SNP by following the same attack steps for SEV-ES, it seems step ① to ④ would work the same. As the VMSA is also protected by the RMP, loading VMSA would lead to an RMP check. However, as the attacker VM uses the victim's ASID, the check would pass. However, the NPF in step ⑤ that reveals the page content would not occur. Instead, an NPF due to RMP check would take place, because the gPA used in nPT walk is different from the one stored in the RMP entry. Therefore, from the description of the RMP, it seems CROSSLINE V1 can be prevented.
- **CROSSLINE V2 on SEV-SNP.** As CROSSLINE V2 does not work on SEV-ES, it cannot be applied on SEV-SNP.

Nevertheless, SEV-SNP is still in its planning phase. Some implementation details are still unclear[4]. For instance, in our discussion with AMD engineers, AMD is developing technologies to better isolate TLBs [6], which will thwart the attack variant we discuss in Section 4.4.1. But it is not yet clear when the technology can be officially announced and implemented on SEV-SNP processors.

4.4.3 Intel MKTME

Similar to AMD's SEV, Intel's Total Memory Encryption (TME) and Multi-Key Total Memory Encryption (MKTME) [42] also provide memory encryption to software. The concept of TME is similar to AMD SME: a memory encryption engine is placed between the

[4]"This white paper is a technical explanation of what the discussed technology has been designed to accomplish. The actual technology or feature(s) in the resultant products may differ or may not meet these aspirations. Each description of the technology must be interpreted as a goal that AMD strived to achieve and not interpreted to mean that any such performance is guaranteed to be fully achieved." [8].

direct data path and external memory buses, which encrypts data entering or leaving the SOC using 128-bit AES encryption in the XTS mode of operation. MKTME is built atop TME and supports multiple encryption keys. When used in virtualization scenarios, MKTME is close to AMD's SEV. However, different from SEV, where each VM only possesses one encryption key, multiple keys can be used in each VM on an MKTME platform, allowing cross-VM memory sharing when the same keys are used. The selection of encryption keys is controlled by software, by specifying the key id in the upper bits of a page table entry (PTE). In a virtualization scenario, the hypervisor has to be trusted because it has the capability of mapping guest VM's memory and controlling the memory encryption keys in the PTE. CROSSLINE is not needed in the MKTME setting as a malicious hypervisor may directly read encrypted guest memory. Therefore, the hypervisor is included in the TCB of MKTME, which could greatly limit its real-world adoption.

4.4.4 Relation to Speculative Execution Attacks

CROSSLINE is *not* a speculative execution attack. Meltdown [64], Spectre [52], L1TF [87], and MDS [25, 76, 89] are prominent speculative execution attacks that exploit transiently executed instructions to extract secret memory data through side channels. In these attacks, instructions are speculatively executed while the processor awaits resolution of branch targets, detection of exceptions, disambiguation of load/store addresses, *etc.*. However, in the settings of CROSSLINE V1, no instructions are executed, as the exceptions take place as soon as the frontend starts to fetch instructions from the memory. The other two variants of CROSSLINE execute instructions with architecture-visible effects.

CROSSLINE does not rely on micro-architectural side channels, either. Speculative execution attacks leverage micro-architectural side channels (*e.g.*, cache side channels) to

leak secret information to the program controlled by the attacker. In contrast, CROSSLINE reveals data from the victim VM as page frame numbers, which can be learned by the hypervisor directly during page fault handling.

4.5 Summary

This chapter demystifies AMD SEV's ASID-based isolation for encrypted memory pages, cache lines, and TLB entries. For the first time, it challenges the "security-by-crash" design philosophy taken by AMD. It also proposes the CROSSLINE attacks, a novel class of attacks against SEV that allow the adversary to launch an attacker VM and change its ASID to that of the victim VM to impersonate the victim. Two variants of CROSSLINE attacks have been presented and successfully demonstrated on SEV machines. They are the first SEV attacks that do not rely on SEV's memory integrity flaws.

Chapter 5: TLB Poisoning Attacks on AMD Secure Encrypted Virtualization

In this chapter, we provide the first exploration of the security issues of TLB management on SEV processors and demonstrate a novel class of TLB Poisoning attacks against SEV VMs. We first demystify how SEV extends the TLB implementation atop AMD Virtualization (AMD-V) and show that the TLB management is no longer secure under SEV's threat model, which allows the hypervisor to poison TLB entries between two processes of a SEV VM. We then present TLB Poisoning Attacks, a class of attacks that break the integrity and confidentiality of the SEV VM by poisoning its TLB entries. Two variants of TLB Poisoning Attacks are described in the chapter; and two end-to-end attacks are performed successfully on both AMD SEV and SEV-ES.

Responsible disclosure. We have disclosed the vulnerability that enables TLB Poisoning Attacks to AMD via emails in December 2019. After an in-depth teleconference discussion with the SEV team, we have been confirmed that the vulnerability exists on SEV and SEV processors, but the upcoming SEV-SNP has a new feature that prevents the attack. Therefore, AMD will not release a patch for the discovered vulnerability but will rely on the new SEV-SNP processor as a line of defense.

5.1 Background

In this section, we present some additional background information about SEV's memory and TLB isolation.

Secure Encrypted Virtualization (SEV). As AMD's new memory encryption feature for AMD-V [12], SEV aims to produce a confidential VM environment in the public cloud and protect VMs from the privileged but untrustworthy cloud host (*e.g.*, the hypervisor). SEV is built atop an on-chip encryption system composed of an ARM Cortex-A5 co-processor [51] and AES encryption engines. The co-processor, also known as AMD-SP, stores and maintains a set of VM encryption keys (K_{vek}) which is uniquely assigned to each SEV-enabled VM. The K_{vek} in the co-processor could not be accessed by either the privileged hypervisor or the guest VM itself. The AES encryption engine automatically encrypts all data in the memory, and decrypts them in the CPU by using the correct K_{vek}.

Nested Page Tables. AMD adopts two-level of page tables to help the hypervisor manage the SEV VM's memory mapping. The upper-level page table, also called the guest page table (gPT), is part of the guest VM's encrypted memory and is maintained by the guest VM, and is usually a 4-level page table that translates the guest virtual address (gVA) to the guest physical address (gPA). Moreover, Guest Page Fault (gPF) caused by the gPT walk is trapped and handled by the guest VM. The lower-level page table is also called NPT or host page table (hPT), which translates gPA to system physical address (sPA), and is maintained by the hypervisor. The NPT structure gives the SEV VM the ability to configure the memory pages' encryption states. By changing the C-bit (Bit 47 in the page table entry) to be 1 or 0, the states of the guest VM's memory page can either be private (encrypted with his K_{vek}) or

shared (encrypted with the hypervisor's K_{vek}). The gPT and all instruction pages are forced to be private states no matter of the value of C-bit.

Moreover, Nested Page Faults (NPF) may be triggered by the hardware during the NPT walk. According to the NPF event, the hypervisor can grab useful information that could reflect the behavior of a program, and therefore leak sensitive information, including the gPA of the NPT and the NPF error code [6]. This forms a well-known controlled-channel attack [39, 61, 91], which compromises SEV's confidentiality and integrity.

Address Space Layout Randomization (ASLR). ASLR is a widely used spectrum protection technique that randomizes the virtual memory areas of a process to defend against memory corruption attacks. This defense mechanism prevents attackers from directly learning the pointer's virtual address and forces them to rely on software vulnerabilities or side-channel attacks [16, 40, 46, 54] to locate the randomized virtual address. Different operating systems have different ASLR implementations. For example, a 64-bit Linux system usually exhibits 28-bit of ASLR entropy for executable [35] while Windows 10 exhibits only 17-19 bits of ASLR entropy for executables [92].

Translation Lookaside Buffer (TLB) and Address Space Identifier (ASID). TLB is a caching hardware inside the chip's memory-management unit (MMU). After a successful page table walk, the mapping from the virtual address to the system address is cached in TLB. For a nested page table on SEV, the mapping of the gVA and the sPA is cached in the TLB. During a page table walk, given a guest CR3 (gCR3) and a host CR3 (hCR3), the hardware automatically translate a gVA to a sPA using the two-level page tables despite the gPT and the NPT are encrypted by different K_{vek}s. AMD-SP uses ASID to uniquely identify the SEV-enabled VM and its K_{vek}. ASID is also part of the tag for both cache lines and TLB entries [51].

5.2 Understanding and Demystifying SEV's TLB Isolation Mechanisms

In this section, we briefly sketch our understanding of TLB isolation mechanisms used in AMD Virtualization for both non-SEV VMs and SEV-enabled VMs. For some of the mechanisms that are not documented, we experimentally validated our conjectures.

5.2.1 TLB Management for Non-SEV VMs

To avoid frequent TLB flushes during VM world switches, AMD introduced ASID in TLB entries [3]. ASID 0 is reserved for the hypervisor and the rest of the ASID are used by the VM. The range of the ASID pool can be determined by CPUID 0x8000000a[EBX]. TLB is tagged with the ASIDs of each VM and the hypervisor, which avoids flushing the entire TLB at the world switch and also prevents misuses of the TLB entries belonging to other entities.

We explore the TLB management algorithm for non-SEV VMs by diving into the source code of AMD SVM [9]. Specifically, the hypervisor is responsible for maintaining the uniqueness and the freshness of the ASID in each logical core of the machine. For each logical core, the hypervisor stores the most recently used ASID in the `svm_cpu_data` data structure. Before each VMRUN of a vCPU of a non-SEV VM, the hypervisor checks whether the CPU affinity of the vCPU has changed by comparing the ASID stored in its VMCB with the most recently used ASID of this logical core. If a mismatch is observed, which means either the vCPU was not running on this logical core before the current VMEXIT or more than one vCPUs sharing the same logical core concurrently, the hypervisor assigns an incremental and unused ASID to this vCPU. In either of these cases, the increment of the

ASID ensures the residual TLB entries cannot be reused. Otherwise, no TLB flushing is needed and the vCPU can keep its ASID and reuse its TLB entries after VMRUN.

The hypervisor is in charge of enforcing TLB flushes under certain conditions. For example, when the recently used ASID exceeds the max ASID range on the logical core, a complete TLB flush for all ASIDs is required. To flush TLBs, the hypervisor sets the TLB_CONTROL bits in TLB_CONTROL filed (058h) of the VMCB during VMEXITs. With different values of bits 39:32 of TLB_CONTROL, the hardware will perform the different operation on the TLB:

- TLB_CONTROL_DO_NOTHING (00h). The hardware does nothing.
- TLB_CONTROL_FLUSH_ALL_ASID (01h). The hardware flushes the entire TLB.
- TLB_CONTROL_FLUSH_ASID (03h). The hardware flushes all TLB entries whose ASID is equal to the ASID in the VMCB.
- TLB_CONTROL_FLUSH_ASID_LOCAL (07h). The hardware flushes this guest VM's non-global TLB entries.
- Other values. All other values are reserved, so other values may cause problems when resuming guest VMs.

After each VMRUN, hardware checks these bits and performs the corresponding actions. The hypervisor is in charge of informing the hardware to flushes TLBs and maintain TLB isolation. Hardware may also automatically perform a partial TLB flush without triggering a special VMEXIT when observing context switches or MOV-to-CR3 instructions. In such cases, only the TLB entries tagged with the current ASID (either in guest ASID or the hypervisor ASID) are flushed [6].

5.2.2 Demystifying SEV's TLB management

The TLB management for SEV VMs and non-SEV VMs is slightly different. The ASIDs of SEV VMs remain the same in their lifetime. Therefore, instead of dynamically assigning an ASID to a vCPU, all vCPUs of the same SEV VM have the same ASID. At runtime, TLB flush is still controlled by the hypervisor. Especially, KVM records the last resident CPU core of each vCPU. For each CPU logical core, it also records the VMCB of the last running vCPU (*sev_vmcbs*[*asid*]) for each ASID. Before the hypervisor resumes a vCPU via VMRUN, it sets the TLB control field in the VMCB to the value of TLB_CONTROL_FLUSH_ASID when (1) this vCPU was not run on this core before or (2) the last VMCB running on this core with the same ASID is not the current VMCB. This enforces the isolation between two vCPUs of the same SEV VM. The code is listed in Listing 5.1. However, if the hypervisor chooses not to set the TLB control field, no TLB entries will be flushed.

```
1   struct svm_cpu_data *sd = per_cpu(svm_data, cpu);
2   int asid = sev_get_asid(svm->vcpu.kvm);
3   pre_sev_es_run(svm);
4   svm->vmcb->control.asid = asid;
5 // No CPU affinity change and No VMCB change
6   if (sd->sev_vmcbs[asid] == svm->vmcb &&
7       svm->vcpu.arch.last_vmentry_cpu == cpu)
8     return;
9 //Otherwise, flush the TLB tagged with the ASID
10  sd->sev_vmcbs[asid] = svm->vmcb;
11  svm->vmcb->control.tlb_ctl = TLB_CONTROL_FLUSH_ASID;
12  vmcb_mark_dirty(svm->vmcb, VMCB_ASID);
13 }
```

Listing 5.1: Code snippet of `pre_sev_run()`.

Experiments to demystify TLB tags. According to AMD manual [6], ASID is part of TLB tag. But is unclear what are the remaining parts of the tag. We conducted some experiments to explore the structure of TLB tags. Specifically, we checked whether vCPUs' TLB entries on the co-resident logical cores will influence each other and whether TLB

entries from the different VM modes (non-SEV, SEV, or SEV-ES) will influence each other. The experiment settings are similar. To explore TLB isolation between co-resident logical cores, we manually set the ASID of two vCPUs to the two co-resident logical cores of the same physical core. To explore TLB isolation between VMs with different VM modes (*e.g.*, SEV and non-SEV), we configured a non-SEV VM and a SEV/SEV-ES VM on the same logical core and set the non-SEV VM's ASID to be identical to the SEV/SEV-ES VM's ASID. In both cases, we skipped the TLB flush to check whether the TLB poison is observed (using steps in Section 5.3.2.1). In neither of two cases, TLB poison is observed. Therefore, we conclude:

- **ASID.** ASID is part of the TLB tag, which provides TLB isolation for TLB entries with different ASID.
- **Logical Core ID.** The Logical Core ID is also part of the TLB tag, which provides TLB isolation for TLB entries on the same physical core but different logical cores.
- **VM mode.** VM mode is part of the TLB tag. Even a non-SEV VM may have the same ASID as a SEV or SEV-ES VM, however, the TLB tag field contains information about the VM's mode, which isolates TLB entries from VMs under different modes.

Besides these components, we have also conjectured that C-bits—the C-bit in the guest page table (gC-bit) and the C-bit in the nested page table (nC-bit)—are also part of the TLB tag. The reason is that when address translation bypasses the page table walk, the values of the gC-bit and nC-bit are still required for the processor to determine which ASID to present to AMD-SP if memory encryption is needed. However, there is no direct evidence for us to conclude the exact C-bit tag format in TLB entries. We have no way to empirically affirm that, for instance, whether both of the C-bits are in the TLB tag or only one C-bit is in the TLB tag.

Table 5.1: TLB flush rules. The *World* column indicates whether the event happens in host world or the guest world; *TLB tag* represents the TLB entry's ASID to be flushed—the host's ASID is 0 and the SEV VM's ASID is N; *Forced* indicates whether the TLB flush is forced by the hardware or controllable by the hypervisor. * highlights a special case, in which when the world switch happens between two vCPUs, the TLB tagged with 0 is forced to be flushed while the TLB tagged with N is flushed under the control of the hypervisor.

World	Events	TLB Tag	Forced
Host/Guest	MOV-to-CR3, Context-switch	0/N	✓
Host/Guest	Update Cr0.PG	0/N	✓
Host/Guest	Update CR4 (PGE, PAEm and PSE)	0/N	✓
Host	Address transliation Registers	All	✓
Host	Activate an ASID for SEV VM	N	✓
Host	Deactivate an ASID for SEV VM	N	✗
Host	ASID exceeds ASID pool range	All	✗
Host	Two vCPUs switch	0+N*	✓+✗*
Host	Change vCPU's CPU affinity	N	✗

5.2.3 TLB Flush Rules for SEV VMs

We summarize the TLB flush rules for SEV/SEV-ES VMs in both hardware-enforced TLB flush and the hypervisor-coordinated TLB flush in Table 5.1. The hardware-enforced TLB flush rules cannot be skipped, while the hypervisor-coordinated TLB flush can be skipped by a malicious hypervisor, which is the root cause of the TLB Poisoning Attack.

Hardware-enforced TLB flushes. All TLB entries are flushed when there is System Management Interrupt (SMI), Returning from System Management (RSM), Memory-Type Range Register (MTRR), and I/O Range Registers (IORR) modifications or MSRs access related to address translation, no matter their ASIDs. At the same time, hardware will automatically flush TLB tagged with the current ASID when observing activities like MOV-to-CR3, context switches, updates of CR0.PG, CR4.PGE, CR4.PAEm and CR4.PSE. Hardware will also force a TLB flush when the hypervisor wants to activate an ASID for a SEV VM.

Hypervisor-coordinated TLB flushes. There are mainly two cases where the hypervisor is coordinated in TLB management. (1) When different VMCB with the same ASID (different vCPUs of the same SEV VM) is to be run on the same logical core. (2) The VMCB to be run was executed on a different logical core prior to this VMRUN.

5.3 Attack Primitives

In this section, we discuss the threat models consider in this paper, and then introduce three attack primitives: TLB misuse across vCPUs (Section 5.3.2), TLB misuse within the same vCPU (Section 5.3.3), and a covert data transmission channel between the hypervisor and a process in the victim VM that is under the adversary's control (Section 5.3.4).

5.3.1 Threat Model

We consider a scenario where the platform is hosted by a hypervisor controlled by the adversary. The victim VM is a SEV-ES enabled VM and thus protected by all SEV-ES features. We assume the ASLR is enabled inside the victim VM.

There is an unprivileged attacker process controlled by the adversary running in the victim VM. The attacker process does not have access to the kernel or learn sensitive information from procfs. The attacker process does not need to have capabilities to perform network communication. We note that the assumption of having an attacker process running inside the victim VM can be weakened (see Section 5.5). The victim process can be any process in the victim VM other than the attacker processes. We assume the adversary can learn the virtual address range of the victim VM via other attacks, such as CrossLine attacks [60].

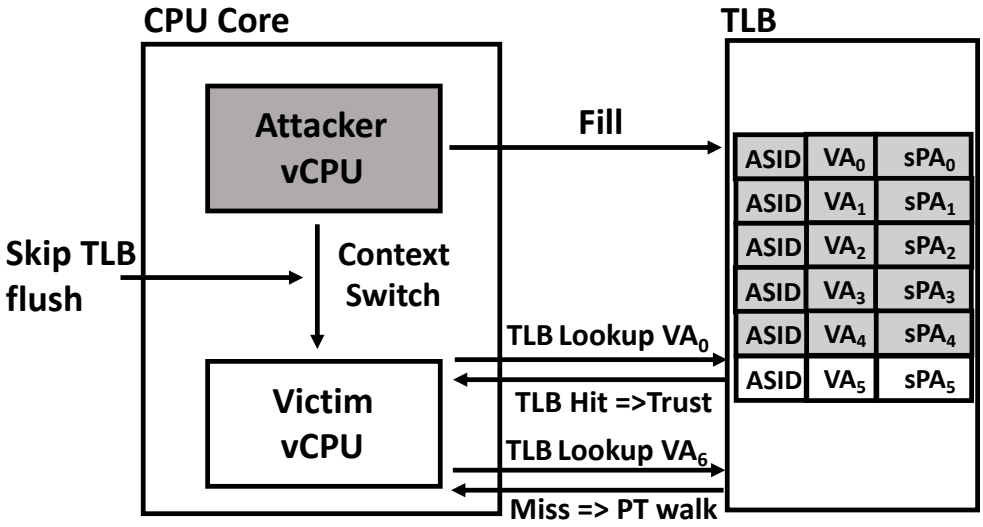


Figure 5.1: TLB misuses across vCPUs.

5.3.2 TLB Misuse across vCPUs

When the victim VM has more than one vCPU, the attacker process and the victim process can run on different vCPUs. We call the vCPU running the attacker process the attacker vCPU and the vCPU running the victim process the victim vCPU. The adversary can misuse TLB entries by skipping the TLB flush during the context switch of these two vCPUs. We use two examples to show how this may be exploited to breach the integrity and the confidentiality of the victim process.

5.3.2.1 TLB Poisoning

We first show that by poisoning TLB entries, the attacker process can alter the execution of the victim process. The attack is illustrated in Figure 5.1.

- **Step-I**: The victim process is suspended before executing an instruction at address VA_0. This can be achieved by manipulating PTEs to trigger NPFs. Note that the content of this instruction is not relevant to this attack.

- **Step-II**: The hypervisor schedules the attacker vCPU to the same logical core as the victim vCPU, and the TLB control field is set to TLB_CONTROL_FLUSH_ASID (03h) to flush the TLB entries with the SEV VM's ASID.
- **Step-III**: It then instructs the attacker process to run an instruction sequence "mov $0x2021, %rax; CPUID" also at address VA_0. The CPUID instruction will trigger a VMEXIT. During the VMEXIT, the attacker vCPU is paused, and the victim vCPU is scheduled to run without flushing the TLB entries.
- **Step-IV**: When the victim process executes the instruction at VA_0, a VMEXIT due to CPUID can be observed with the %rax value set to 0x2021 in the GHCB. This means the victim process has been successfully tricked to execute the same instruction as the attacker process at VA_0, because it reuses the TLB entry poisoned by the attacker process.

5.3.2.2 Secret Leaking

The second example shows that the attacker process can read the victim process's memory space directly.

- **Step-I**: The attacker process uses `mmap()` syscall to pre-map a data page such that the virtual address VA_0 points to a data region on this page.
- **Step-II**: The victim process is scheduled to run and accesses the memory at address VA_0, which can be either a instruction fetch or a data load. This step loads a TLB entry into the TLB.
- **Step-III**: The victim vCPU is de-scheduled by the hypervisor, and the attacker vCPU is scheduled to run on the same logical core. The hypervisor sets the TLB control field of the attacker's VMCB to TLB_CONTROL_DO_NOTHING (00h), such that no TLB entry is flushed.

- **Step-IV**: After being scheduled to run and loading data from VA_0, we observe that the attacker process successfully loads the data from the victim's address space, compromising the victim's confidentiality. This is because the TLB entries created by the victim process is reused by the attacker process.

5.3.3 TLB Misuse within the Same vCPU

When the victim VM has only one vCPU, the attacker process shares the vCPU with the victim process. In this case, TLB misuse is less straightforward. The TLB flush rules we illustrated in Section 5.2.3 suggest that the hardware will automatically flush the entire TLB tagged by the victim VM vCPU's ASID when there is an internal context switch in the guest VM, which leaves no chance for the hypervisor to skip the TLB flush. As such, the hypervisor cannot directly misuse the TLB entries between two processes within the same vCPU. To address this challenge, we propose a novel VMCB-switching approach to bypass the hardware-enforced TLB flush during the internal context switch.

5.3.3.1 Bypassing Hardware-enforced TLB Flushes

The key to bypassing the hardware-enforced TLB flush is to reserve the attacker process's TLB entries on one CPU core and then migrate the vCPU to another CPU core. The internal context switch between the victim process and the attacker process is then performed on the second CPU core, which automatically flushes all TLB entries on the second logical core. Because the hypervisor isolates the first CPU core to prevent other processes from evicting its TLB entries, the TLB entries of the attacker processes are hence preserved. The hypervisor then migrates the vCPU back, with the victim process executing on it. The victim process will then misuse the TLB entries *poisoned* by the attacker process.

The challenges for bypassing the hardware-enforced TLB flush are two-fold: First, changing the vCPU affinity inside the victim VM leads to TLB flush for both the victim and attacker processes, which, nevertheless, can only be done by a privileged process. Secondly, changing the CPU affinity outside the victim VM—from the hypervisor side—may easily evict the reserved TLB entries. Thus, traditional CPU schedule methods like `taskset` or `sched_setaffinity` cannot work in our case.

5.3.3.2 VMCB Switching

The following VMCB-switching approach can be used to bypass the hardware-enforced TLB flushes (shown in Figure 5.2).

- **Step-I:** The hypervisor first isolates the target vCPU hosted in a hypervisor process HP_1 on logical core LC_1 and prevents other processes from accessing LC_1, as well as its co-resident logical core on the same physical core. The hypervisor also reserves another logical core LC_2 with an idle hypervisor process HP_2. This is to ensure irrelevant processes will not evict the reserved TLB entries.
- **Step-II:** After the attacker process poisons the targeted TLB entries, the hypervisor traps the vCPU into a yield() loop during one VMEXIT. Meanwhile, the hypervisor lets the idle process HP_2 on LC_2 to resume the attacker vCPU using its VMCB, VMSA pointer, and NPT structures. This is possible because all states of the attacker vCPU (*e.g.*, registers, ASID, Nested CR3) are stored in the DRAM, encrypted using either hypervisor's memory encryption key (*e.g.*, VMCB, NPT) or the guest VM's VM encryption key (*e.g.*, VMSA). After resuming the attacker vCPU on LC_2, there are no valid TLB entries on LC_2, but the attacker process inside the attacker vCPU can continue execution after page table walks.

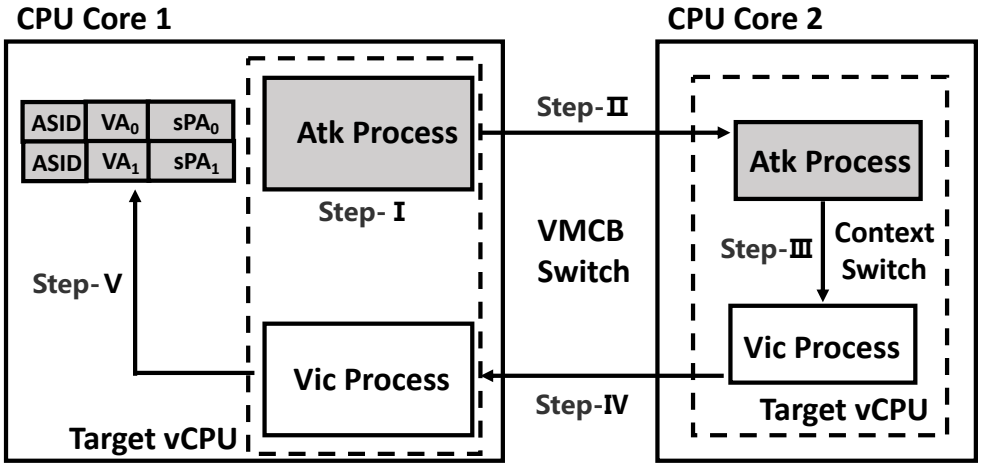
- **Step-III:** The hypervisor traps and traces gCR3 changes to monitor the internal context switches on the attacker vCPU. Specifically, it intercepts TRAP_CR3_WRITE VMEXIT and extract the gCR3 value in the EXITINFO1 field of VMCB. Since the inner context switch happens on LC_2, no hardware-enforced TLB flush is triggered on LC_1, and thus the attacker process's TLB entries are preserved on LC_1.
- **Step-IV:** After observing a context switch from the attacker process to the victim process is scheduled, the hypervisor switches the attacker vCPU back to LC_1 following a similar method described in Step-II. The hypervisor stops HP_2 on LC_2 and releases HP_1 on LC_1 from the empty loop.
- **Step-V:** After resuming execution on LC_1, the victim process first tries to execute its next instruction pointed by `RIP` in VMSA via a TLB lookup. The preserved TLB entries on LC_1 are unconditionally trusted by the hardware. After the victim process has used the attacker's TLB entries to execute instructions, some remaining TLB entries belonging to the attacker process may potentially disturb the execution of the victim process afterwards. Thus, the hypervisor can choose to perform a total TLB flush.

Note that the attacker process and the hypervisor can also breach the confidentiality of the victim process in a reversed way, where the hypervisor reserves the victim process's TLB entries and let the attacker process to reuse it to exfiltrate secrets from the victim's address space.

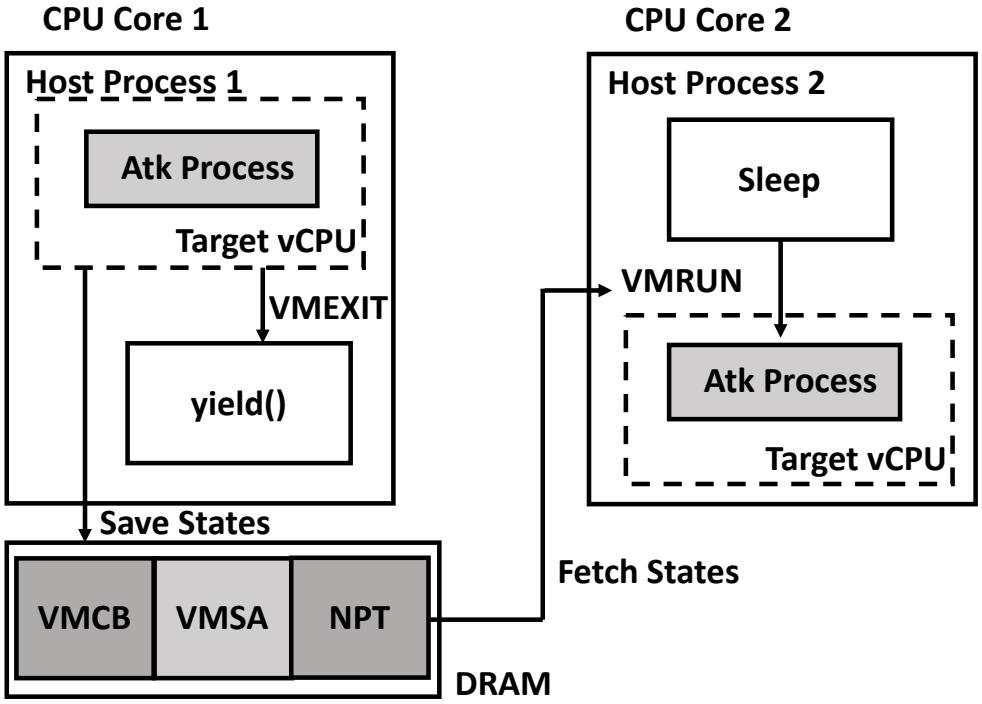
5.3.4 CPUID-based Covert Channel

The third primitive we build is for transmitting data between the hypervisor and the attacker process in the victim VM that is under the adversary's control. To do so, we build a CPUID-based covert channel so that network communication is not required. The

adversary-controlled process may execute CPUID instructions to receive data or pass the data to the hypervisor. Specifically, to send data to the hypervisor, the attacker process may trigger a CPUID with a reserved RAX value (*e.g.*, 1234) to initiate data transfer. The attacker process then repeatedly triggers CPUID with RAX filled with the data to be transferred. Similarly, to receive data from the hypervisor, the attacker process can trigger a CPUID with another reserved RAX value (*e.g.*, 1235). The hypervisor retrieves the value of RAX and passes the data into GHCB's RAX field before VMRUN. The attacker process can then read the value of RAX after the CPUID instruction. Data received from the covert channel can use used as commands; the attacker process performs pre-defined actions (*e.g.*, mmap memory page and read certain virtual address) in accordance with the command received. On our testbed, the maximum transmission speed is 1.854MB/s when using the 8-byte RAX register for data transmission. Other covert channels that make use of cache timing [65, 66] or AMD's way predictor [63] can also be adopted as covert channels, but are less robust.



(a) VMCB switching steps.



(b) **Step-II:** Change vCPU's CPU affinity without TLB flush.

Figure 5.2: VMCB switching.

5.4 TLB Poisoning with Assisting Processes

In this section, we introduce the first variant of TLB Poisoning attacks, which is assisted by an unprivileged attacker process running in the victim VM. Following the threat model described in Section 5.3, we assume the attacker process is unprivileged with limited access to system resources, such as procfs, networking, or any privileged system capabilities. This is practical either when the adversary has an unprivileged user account on the victim VM or an application with security vulnerabilities remotely exploitable by the adversary. To simplify the attack, we assume the ASLR is disabled on the victim VM or the attacker process can learn the virtual memory area (VMA) of the victim process. In a real attack, the attacker process can break the ASLR either by CROSSLINE attack or other existing methods [16, 40, 60].

5.4.1 Case Study: OpenSSH

In this case study, we show that with the help of an unprivileged attacker process within a guest VM, the adversary can poison the TLB entries of a privileged victim process and then control its execution. The attack is applied to OpenSSH and used to bypass password authentication.

5.4.1.1 OpenSSH's Process Management

The sshd daemon process (denoted P_d) is launched during system boot. The daemon process runs in the background and listens to connections on SSH ports (*i.e.*, 22). Its address space is defined in the kernel by the VMA data structures. Upon receiving a connection, P_d forks a sshd child process P_c, which performs a privilege separation (or *privsep*) by spawning another unprivileged process P_n to deal with the network transmission and keeps the root

privilege itself to act as a monitoring process. Once the user has successfully authenticated, P_n is terminated, and a new process P_u is created under the new user's username. In our TLB Poisoning Attack, the victim process is the privileged child sshd process P_c and the attacker process aims to poison the TLB entries of P_c.

5.4.1.2 Password Authentication Bypass

The adversary first initializes a SSH connection to the target VM and monitors gCR3 changes by setting the CR3_WRITE_TRAP intercept bit in its VMCB. When the SSH packet from the adversary is received by the SEV-ES VM, the adversary will immediately observe a context switch (*i.e.*, gCR3 change). The new process to run is the sshd child process P_c. In this way, the adversary can identify the gCR3 of P_c.

① **Locate the shared library.** The attacker process first helps the adversary to locate the gPA of the shared library. In our attack, we target at `pam_authenticate()`, which is a function of the shared library `libpam.so.0` and used by sshd for password authentication. `pam_authenticate()` returns 0 if the authentication succeeds. The adversary can use the attacker process to help locate the gPA of `pam_authenticate()` (denoted gPA_{pam}). He first synchronizes with the two colluding entities using the covert channel described in Section 5.3.4 and then calls `pam_authenticate()` from the attacker process. The hypervisor can learn gPA_{pam} by triggering NPFs.

② **Track the victim's execution.** The adversary clears the Present bit of all pages and monitors NPFs after intercepting his SSH packet with the incorrect password. If a NPF of gPA_{pam} is observed, the adversary knows the victim process is going to authenticate the password by calling `pam_authenticate()`. The adversary then pauses the victim process by trapping the victim in the gPA_{pam} NPF handler. This is used to provide a time window

for the attacker process to poison the TLB entries. Note that this step is rather important in real attacks. The attacker process needs to poison the TLB entries right before the victim process accessing those poisoned TLB entries. Otherwise, the poisoned TLB entries may be evicted by other activities.

③ **Poison TLB entries.** The adversary can then poison the TLB entries of the victim. Let the virtual address of the instruction page containing `pam_authenticate()` in P_c be gVA_{pam}. We assume the adversary can learn gVA_{pam} in advance. gVA_{pam} is predictable if ASLR is disabled. The adversary can also learn gVA_{pam} using existing attack methods [16, 40, 60]. The adversary targets at poisoning the TLB entries indexed by gVA_{pam}. Specifically, the attacker process first mmap a page with the virtual address to be gVA_{pam}. Note that gVA_{pam} is only used in P_c and the attacker process can assign this virtual address to a new instruction page. The attacker process then copies the same instruction page as the victim into the new page, but replaces a few instructions of `pam_authenticate` (offset $0x5b0$ - $0x65f$ of the binary, starting with `test %rdi %rdi`) with `mov $0 %eax` and `ret` ($0xb8$ $0x00$ $0x00$ $0x00$ $0x00$ $0xc3$). The adversary also schedules the attacker process to the same logical core as the victim process by changing the CPU affinity of the vCPU. The attacker process then repeatedly accesses this instruction page in a loop to preserved the TLB entries.

④ **Bypass authentication.** After the attacker process poisons the TLB entries of `pam_authenticate()`, the adversary directly resumes P_c without a TLB flush. Recall in step ②, P_c was paused before a page table walk to resolve gPA_{pam}. The adversary resumes P_c without handling this page table walk in order to force P_c to reuse the poisoned TLB entries. In this way, when P_c calls `pam_authenticate()`, it will execute the instruction in the attacker's address space. Therefore, the function will directly return with an 0 in `EAX` and thus allow arbitrary user to login.

5.4.2 Evaluation

The experiment settings are list below. The CPU we used is AMD EPYC 7251 with 8 physical cores. All the software needed to launch a SEV-ES VM is download from AMD SEV repository [9]. The host kernel version is sev-es-v3 . The QEMU version used was sev-es-v12 and the OVMF version was sev-es-v27 . The victim VM was a SEV-ES-enabled VMs with 4 vCPUs, 4 GB DRAM and 30 GB disk storage. The OpenSSH version is OpenSSH_7.6p1 and the OpenSSL version is 1.0.2n. We repeated the attack 20 times and evaluated the attacks in terms of successful rate: All the 20 attacks could successfully bypass the password authentication and logged in with incorrect passwords.

5.5 TLB Poisoning without Assisting Processes

In this section, we show that TLB Poisoning attacks can work even without the help of an attacker process in the victim VM. The intuition is that when processes share similar virtual address spaces, TLB misuse may happen between these processes without direct control of either of them.

Specifically, we target at `fork()`, which is a system call used to create new processes. `fork()` is widely used in server-side applications, *e.g.*, OpenSSH, sftp, Nginx, and Apache web server, to serve requests from different clients. The forked child processes has a high probability to share a very similar virtual memory area with majority of their virtual address space layout overlapped. Even the VM's administrator chooses to enable ASLR, the same VMA randomization will be applied to the parent process and all child processes, which gives the adversary the chance to conduct TLB poisoning without concerning about the unpredictable VMA. This similarity of address spaces of forked processes has been exploited in memory hijacking attacks [56].

Attack scenarios. Similar to the previous case study, we choose to showcase our TLB Poisoning attack against an SSH server. But this time, we target Dropbear SSH [48], which is a lightweight open-source SSH server written in C and released frequently since 2003. We did not choose the more popular OpenSSH because it alters its memory address space in all its children processes that serve incoming connections (by calling `exec()`). However, this mechanism is only observed in OpenSSH and OpenBSD. Other network applications like Dropbear SSH and Nginx will not change their virtual memory layout for different connections.

We assume the targeted Dropbear SSH server application is free of memory safety vulnerabilities and timing channel vulnerabilities. We assume the binary of the Dropbear Server application is known by the adversary. We assume the username of a legitimate user is also known by the adversary; this is a practical assumption as usernames are not considered secrets. To simplify the attack, we also assume the two processes are scheduled on two different vCPUs, which makes the attack easier to perform; otherwise the VMCB-switching approach is required.

5.5.1 Poison TLB Entries between Connections

We consider two SSH connections: One is the connection from the adversary, which is served by the process P_{atk} that is forked from the DropSSH server daemon; the other is a connection from a legitimate user, which is served by the process P_{vic}. The attack goal is to allow the attacker process to temporarily use the victim process's TLB entries and circumvent the password authentication.

Regular login procedures. After the login password packet is received by the victim VM, P_{vic} calls `svr_auth_password()` to validate the password. As shown in Listing 5.2,

the password encryption function in the POSIX C library `crypt()` is called to generate a hash of the user-provided password. The result is stored in a buffer called `testcrypt`. The buffer storing the plaintext of the password is freed immediately. After that, the hash of the user-provided password is compared with the stored value in the system file using `constant_time_strcmp()`, which returns 0 if these two strings are identical. If the user-provided password is correct, P_{vic} will take the correct-password branch, which calls `send_msg_userauth_success()`. Otherwise, the incorrect-password branch is taken.

```
void svr_auth_password(int valid_user) {
   char * passwdcrypt = NULL;
   // store the crypt from /etc/passwd
   char * testcrypt = NULL;
   // store the crypt generated from the password sent
   ...
   // **Execution Point 1 (NPF)
   if (constant_time_strcmp(testcrypt, passwdcrypt) == 0) {
   // successful authentication
   // **Execution Point 2 (NPF)
       send_msg_userauth_success();
      } ...
}
```

Listing 5.2: Code snippet of `svr_auth_password()`.

Attack overview. We show that by breaking the TLB isolation, the attacker process P_{atk} can bypass the password authentication even with an incorrect password. Specifically, the virtual addresses of the `testcrypt` buffer are usually the same for both P_{atk} and P_{vic} (this fact will be empirically evaluated later). We use <gVA$_{pwd}$, sPA$_{vic}$> to denote the TLB entry owned by P_{atk}, which caches the mapping from the virtual address of the `testcrypt` buffer to the system physical address that stores the hashed password used in P_{vic}. The goal here is to make sure the TLB entry <gVA$_{pwd}$, sPA$_{vic}$> is not flushed when P_{atk} executes `constant_time_strcmp()`. In this way, P_{atk} can re-use the `testcrypt` of P_{vic} to circumvent password authentication.

Key challenges. The key challenge in this attack is to ensure only necessary TLB entries are preserved. Otherwise, later TLB entries may flush those necessary TLB entries. To address the challenge, it is important to perform TLB poisoning at the proper execution point. As shown in Figure 5.3, the adversary needs to locate the execution points right before and after the password authentication (*e.g.*, `constant_time_strcmp()`), which can be done using the NPF controlled channels.

The attack overview is shown in Figure 5.3. Let the guest physical address of the instruction page where the `svr_auth_password()` and the `constant_time_strcmp()` are located be gPA_1 and gPA_2, respectively. The adversary first traps the attacker process in an empty loop when handling the NPF of gPA_2 (execution point 1), which means P_{atk} is about to call `constant_time_strcmp()`. Then the adversary will not interrupt P_{vic} until it also reaches the NPF of gPA_2 (execution point 1). When handling this NPF, the adversary triggers a complete TLB flush. P_{vic} then continues execution until it finishes the password authentication and tries to return to `svr_auth_password()`. A NPF of gPA_1 (execution point 2) will be observed and the adversary traps P_{vic}. Meanwhile, the adversary releases the attacker process and skips the TLB flush. All TLB entries used by P_{vic} during the execution of `constant_time_strcmp()` are thus preserved in the TLB, including TLB (gVA_{pwd}, sPA_{vic}). After the attacker process completes `constant_time_strcmp()`, passes the password check, and reaches execution point 2, the adversary triggers a complete TLB flush (to avoid unnecessary TLB misuses) and releases P_{vic}. Both P_{atk} and P_{vic} continue execution as normal afterwards and no traces will be left in the kernel message.

5.5.2 An End-to-end Attack

The adversary follows these steps for an end-to-end attack:

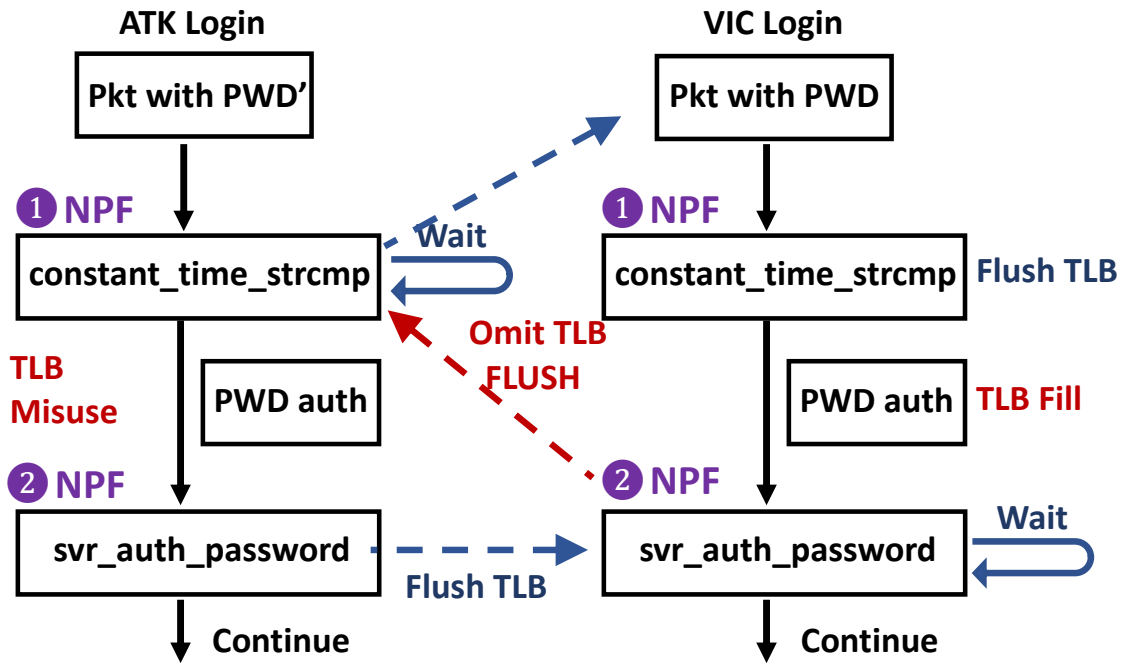


Figure 5.3: Attack steps to bypass password authentication.

① **Monitor network traffic.** Even the adversary cannot directly learn the content of encrypted network packets, the adversary can inspect incoming and outgoing network packets through the unencrypted metadata (*e.g.*, destination address, source address or the port number). The adversary continuously monitors network traffic to identify the SSH handshake procedure. Once the adversary identifies a *client_hello* packet sent from a legitimate user, the adversary traps that packet and sends a *client_hello* packet from a remote machine controlled by himself. Once this *client_hello* packet reaches the victim VM, the adversary resumes the processing of the *client_hello* packet from the legitimate user. Thus, the victim VM shall receive two connection requests, one from the adversary and another from a legitimate user.

② **Monitor fork() and gCR3 changes.** Next, the adversary locates the gCR3 of the forked child processes. During the victim VM's booting period, the adversary continuously monitors gCR3 changes by setting the CR3_WRITE_TRAP intercept bit in the VMCB. Afterwards, gCR3 changes will cause an automatic VMEXIT with the new gCR3 value stored in VMEXIT EXITINFO. After receiving the two SSH connection packets, the Dropbear Daemon will fork twice to generate the child process for the adversary's connection and the legitimate user's connection. We call the forked child process for the adversary's connection P_{atk}, whose gCR3 is gCR3_{atk}. We call the forked child process for the legitimate user's connection P_{vic}, whose gCR3 is gCR3_{vic}. The adversary can identify the gCR3_{atk} and gCR3_{vic} by correlating them with the received *client_hello* packets.

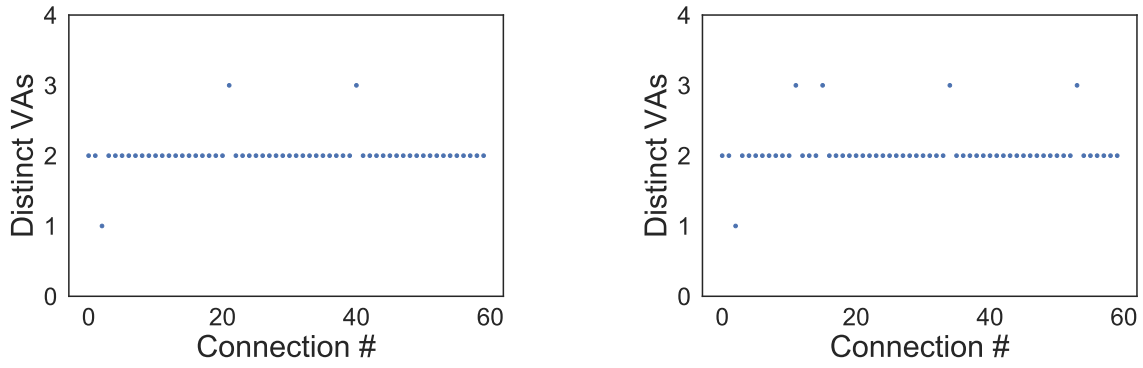
③ **Monitor NPFs to locate the target gPAs.** The adversary may try to log in by sending an arbitrary password. The legitimate user logs in by sending a correct password. The adversary triggers NPFs by clearing the Present bits in the NPT, when the encrypted SSH packets that contain the passwords are observed. A sequence of NPF for P_{atk} and a sequence of NPFs for P_{vic} will be observed. The adversary also collects additional information (*e.g.*, NPF EXITINFO2) along with the NPF VMEXITs, which reveals valuable information. For instance, the adversary can learn that the NPF is caused by write/read access, user/kernel access, code read, or page table walks. The adversary also periodically (*e.g.*, every 50 NPFs) clears all Present bits to fine tune the NPF sequence. Since the Dropbear's binary is known by the adversary, the adversary can learn the NPF patterns offline to locate the gPA of `svr_auth_password()` (denoted gPA_1) and the gPA of the first instruction in `constant_time_strcmp()` (denoted gPA_2). The features used in pattern recognition are the sequence of NPFs and their error code. During the attack, the adversary can use the recognized pattern to locate gPA_1 and gPA_2.

④ **Skip TLB flush.** The adversary continuously monitors P_{atk} and P_{vic}. When the adversary observes the NPF of gPA_2 in P_{atk}, he traps P_{atk} in an empty loop and clears the Present bit of all pages. When the adversary observes the NPF of gPA_2 in P_{vic}, he clears the Present bit for all memory pages and performs a complete TLB flush. The adversary traps P_{vic} when he observes the NPF of gPA_1. P_{atk} is then resumed and the adversary skips the TLB flush. P_{atk} will use the preserved TLB entries from P_{vic} to to read the password hash from the `testcrypto` in the address space of P_{vic}, which leads to a successful login with an incorrect password. To void further TLB pollution, the adversary then forces a complete TLB flush and resumes the victim process. Both P_{atk} and P_{vic} will continue their execution normally afterwards.

5.5.3 Evaluation.

All experiments were performed on a workstation whose CPU is AMD EPYC 7251 Processor (8 physical core with SMT enabled). The VMs (including victim VM and the training VMs) used in this section were SEV-ES-enabled VMs with four vCPUs, 4 GB DRAM, and 30 GB disk storage. The software of the OS, QEMU, and the UEFI image are the same in Section 5.4.2. ASLR is enabled in the SEV-ES-enabled VMs by setting the parameter in /proc/sys/kernel/randomize_va_space to 2. The source code of Dropbear is downloaded from Github [48][5]. The Dropbear SSH Server is configured as the default setting. The Dropbear SSH Server is bond to Port 22. One minor non-default setting to assist the attack is that we forced P_{atk} and P_{vic} to execute on different vCPUs of the victim VM. Note, this setting improves the success rate of the attack but is not necessary in practical attacks.

[5] commit:846d38fe4319c517683ac3df1796b3bc0180be14



(a) 10 connections/s.

(b) 1 connection/s.

Figure 5.4: Variation of the virtual address of `testcrypto`.

Buffer address variation. We first evaluated the variation of the virtual address of `testcrypto` under different connection ratios. In the training VM, the Dropbear server is modified to print the virtual address of `testcrypto` to the console after each connection. Then we used a simple script to initiate new SSH connections, send the correct password to login, obtain the virtual address of `testcrypto`, and end the current SSH connection. In total, 120 connections were collected. For the first 60 connections, the time interval between two contiguous connections was set to 0.1 second. For the second 60 connections, the time interval was set to 1 second. As shown in Figure 5.4a, when the time interval is set to 0.1 second, although 3 different virtual addresses of the `testcrypto` are observed, the virtual address of `testcrypto` remains the same in 57 out of the total 60 connections. When the time interval is set to 1 second, the virtual address of `testcrypto` remains the same in 55 out of the total 60 connections. The experiment results show that the virtual addresses for `testcrypto` are relatively stable for different connections, which gives the adversary the chance to poison the TLB entries of the `testcrypto` buffer between two connections.

Pattern matching. We evaluated the performance of pattern matching. Specifically, we repeated the above attack steps 100 times and performed pattern matching on-the-fly each time. In 98 out of the 100 trials, the adversary is able to correctly recognize the pattern and locate the gPA. The average time used to locate the pattern is 0.10137 second with a standard deviation of 0.02460 second.

End-to-end attacks. We then evaluated the success rate of end-to-end attacks. The adversary conducted end-to-end attacks in the victim VM. An incorrect password is used by the adversary for his SSH connections. The adversary repeated the attacks 20 times. In 17 out of the 20 connections, the adversary is able to log in with the incorrect password. There are two reasons that might count for the 3 failed cases. The first reason is that the reserved TLB entries might be evicted before use. The second reason is that there are false positives in pattern matching. However, the adversary can always repeat the attacks the next time a legitimate user logs in.

5.6 Discussion and Countermeasure

In this section, we discuss applications of TLB Poisoning Attacks on SEV-SNP, their differences compared to known attacks, and their countermeasures.

5.6.1 TLB Poisoning on SEV-SNP

Although we have not tested TLB Poisoning Attacks on SEV-SNP processors, according to the feedback from the AMD team, SEV-SNP has fixed the TLB misuse problem. The latest AMD architecture programmer's manual [6] also shows some newly added fields in the VMSA: TLB_ID (offset 3d0h) and PCPU_ID (offset 3d8h). However, from the public documents, it is unclear how exactly these two fields enforce additional TLB flushes. We

conjecture that the hardware use TLB_ID and PCPU_ID as parts of TLB tags to identify vCPU and TLB entry's ownership. We inspected the source code of software supports of SNP (branch: sev-snp-devel)[6] [9], and failed to locate any software function that controls these two VMCB fields. Therefore, we conjecture these two fields are managed solely by the hardware. The hypervisor can still use TLB_CONTROL field to enforce TLB flushes but has lost the capability to deliberately skipping TLB flushes.

5.6.2 Comparison with Known Attacks

Previous works break the confidentiality and/or the integrity of SEV by replacing unprotected I/O traffic [61], manipulating NPT mapping [68, 69] and unauthenticated encryption [23, 29, 93]. All of these previous works can be mitigated by SEV-SNP via the Reversed Map table (RMP), which establishes a unique mapping between each system physical address with either a guest physical address or a hypervisor physical address. The RMP also records the ownership of each system physical address (*e.g.*, a hypervisor page, a hardware page, or a SEV-SNP VM's page) as well as the ASID. For SEV-SNP VM, the RMP checks the correctness and the ownership after a nested page table walk. Only if the ownership is correct, will the mapping between the guest virtual address and the system physical address be cached in the TLB. This ownership check prevents the hypervisor from remapping the guest physical address to another system physical address and thus prevents attacks that require manipulation of the NPT. Meanwhile, the RMP restricts the hypervisor's ability to write to the guest VM's memory page, which mitigates attacks relying on unauthenticated encryption and unprotected I/O operations.

In contrast, this work is the first to demystify how TLB isolation is performed in SEV and the first to demonstrate the security risks caused by the hypervisor-controlled TLB

[6]Commit: 0965d085cd2453a3512c98924dac70e5cdf17402.

flushes. TLB Poisoning Attacks by themselves do not rely on the known vulnerabilities of SEV and SEV-ES, such as the lack of authenticated memory encryption, the lack of NPT protection, and the lack of I/O protection, and RMP alone does not prevent TLB Poisoning Attacks.

5.6.3 Countermeasures

TLB Poisoning Attacks affect all SEV and SEV-ES servers, including all first and second generation EPYC server CPUs (*i.e.*, Zen 1 and Zen 2 architecture). Older processors may use a microcode patch to enforce a TLB flush during VMRUN for all SEV/SEV-ES vCPUs. From the software side, to mitigate TLB Poisoning Attacks, we recommend all network-related applications (*e.g.*, HTTPS, FTP, and SSH server) to use `exec()` to ensure a completely new address space for a new connection.

5.7 Summary

In this chapter, we present the first work to demystify AMD SEV's insecure TLB management mechanisms and demonstrate end-to-end TLB Poisoning Attacks that exploit the underlying design flaws. Our study not only presents another vulnerability in the design of SEV, but reveals the difficulty of securely isolating TLBs with untrusted privileged software.

Chapter 6: CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel

In this chapter, we study a previously unexplored vulnerability of SEV, including both SEV-ES and SEV-SNP. The vulnerability is dubbed ciphertext side channels, which allows the privileged adversary to infer the guest VM's execution states or recover certain plaintext. To demonstrate the severity of the vulnerability, we present the CIPHERLEAKS attack, which exploits the ciphertext side channel to steal private keys from the constant-time implementation of the RSA and the ECDSA in the latest OpenSSL library.

Responsible disclosure. We disclosed the vulnerability of the ciphertext side channel and the CIPHERLEAKS attack to AMD via emails in December 2020. We also distributed the first draft of this paper with AMD engineers in January 2021. AMD engineers have acknowledged the vulnerability on SEV, SEV-ES, and SEV-SNP, and filed an embargo that is effective until August 10, 2021. As of the time of writing, CVE number, CVE-2020-12966, has been reserved for the vulnerability. AMD will announce a security bulletin together with a hardware patch for SEV-SNP in August 2021.

We have also reported the vulnerable OpenSSL algorithms (see Section 6.3) to OpenSSL in January 2021. The OpenSSL community has acknowledged our notification, but OpenSSL will not be patched, because to properly mitigate such an attack within OpenSSL, it would

require significant changes to the whole software stack. We will describe software countermeasures in Section 6.5.

6.1 Background

6.1.1 Secure Encrypted Virtualization

Secure Encrypted Virtualization (SEV) is a new feature in AMD processors [51]. AMD introduces SEV for protecting virtual machines (VMs) from the untrusted hypervisor. Using the memory encryption technology, each VM will be encrypted with a unique AES encryption key, which is not accessible from the hypervisor or the VMs. The encryption is transparent to both hypervisor and VMs and happens inside dedicated hardware in the on-die memory controller. The in-use data in each VM will be encrypted by their corresponding key automatically, and thus no additional software modifications are needed to run programs containing sensitive secrets in the SEV platform. Open Virtual Machine Firmware (OVMF), the UEFI for x86 VM, and Quick Emulator (QEMU), the device simulator, are the other two critical components for the SEV-enabled VM.

Encrypted Memory. SEV hardware encrypts the VM's memory using 128-bit AES symmetric encryption. The AES engine integrated into the AMD System-on-Chip (SOC) automatically encrypts the data when it is written to the memory and automatically decrypts the data when it is read from memory. For SEV, the AES encryption uses the XOR-and-Encrypt encryption mode [30], which is later changed to an XEX mode encryption. Thus, each aligned 16-byte memory block is encrypted independently. SEV utilizes a physical address-based tweak function $T()$ to prevent the attacker from directly inferring plaintext by comparing 16-byte ciphertext [51]. It adopts a basic Xor-and-Encrypt (XE) mode on the first generation of EPYC processors (*e.g.*, EPYC 7251). The ciphertext c is calculated

by XORing the plaintext m with the tweak function for system physical address P_m using $c = ENC(m \oplus T(P_m))$, where the encryption key is called VM encryption key (K_{vek}). This basic XE encryption mode can be easily reverse-engineered by the adversary as the tweak function vectors t_is are fixed. AMD then replaces the XE mode encryption with the XOR-Encrypt-XOR (XEX) mode in EPYC 7401P processors where the ciphertext is calculated by $c = ENC(m \oplus T(P_m)) \oplus T(P_m)$. The tweak function vectors t_is are proved to have only 32-bit entropy by Wilke *et al.* [93] at first, which allows an adversary to reverse engineer the tweak function vectors. AMD adopted a 128-bit entropy tweak function vectors in their Zen 2 architecture EPYC processors from July 2019 [85] and thus fixed all existing vulnerabilities in SEV AES encryption. However, the same plaintext always has the same ciphertext in system physical address P_m during the lifetime of a guest VM.

SEV, SEV-ES, and SEV-SNP. The first version of SEV [51] was released in April, 2016. AMD later released the second generation SEV-ES [49] in February, 2017 and the whitepaper of the third generation SEV-SNP [50] in January, 2020. SEV-ES is designed to protect the register states during the world switch and introduces the VMSA to store the register states encrypted by K_{vek}. SEV-SNP is designed to protect the integrity of the VM's memory and introduces the RMP to store the ownership of each memory pages. Although SEV, SEV-ES, and SEV-SNP use the same AES encryption engine, some additional memory access restrictions are included in SEV-SNP for integrity protection. In SEV and SEV-ES, the hypervisor has read/write access to the VM's memory regions, which means the hypervisor can directly read or replace the ciphertext of the guest VM. In SEV-SNP, the RMP checks prevent the hypervisor from altering the ciphertext in the guest VM's memory by adding the ownership check before memory accesses. However, the hypervisor still has read accesses to the ciphertext of the guest VM's memory [8].

Non-Automatic VM Exits. VMEXITs in SEV-ES and SEV-SNP are classified as either Automatic VM Exits (AE) or Non-Automatic VM Exits (NAE). AE VMEXITs are events that do not need to expose any register state to the hypervisor. These events include machine check exception, physical interrupt, physical Non-Maskable-Interrupt, physical Init, virtual interrupt, pause instruction, `hlt` instruction, shutdown, write trap of CR[0-15], Nested page fault, invalid guest state, busy bit, and VMGEXIT [6]. All other VMEXITs are classified as NAE VMEXITs, which require exposing some register values to the hypervisor.

Instead of being trapped directly by the hypervisor, NAE events first result in a VC exception, which is handled by a VC handler inside the guest VM. The VC handler then inspects the NAE event's error code and decides which registers need to be exposed to the hypervisor. The VC handler copies those registers' states to a special structure called Guest-Hypervisor Communication Block (GHCB), which is a shared memory region between the guest and the hypervisor. After copying those necessary registers' states to GHCB, the VC handler executes a `VMGEXIT` instruction to trigger an AE VMEXIT. The hypervisor then traps the VMGEXIT VMEXIT, reads those states from the GHCB, handles the VMEXIT, writes the return registers' states into GHCB if needed, and executes a VMRUN. After the VMRUN, the guest VM's execution will resume after the `VMGEXIT` instruction inside the VC handler, which copies the return values from GHCB to the corresponding registers, and then exits the VC handler. For example, to handle `CPUID` instructions, the VC handler stores the states of RAX and RCX and the VM EXITCODE (0x72 for CPUID) into GHCB and executes a `VMGEXIT`. The hypervisor then emulates the CPUID instruction and updates the values of RAX, RBX, RCX, and RDX in GHCB. After VMRUN, the VC handler checks if those return registers' states are valid and copies those states to its internal registers.

IOIO_PROT. During the Pre-Extensible Firmware Interface (PEI) initialization phase of SEV VM, IOIO port is used instead of DMA. The reason is that DMA inside SEV VM requires a shared bounce buffer between VM and the hypervisor [61]. The guest VM needs to copy DMA data from the bounce buffer to its private memory for input data and copy data from its private memory to bounce buffer for output data. Implementing bounce buffer requires allocating dynamic memory and additional memory copy operations, which is a challenge in the PEI initialization phase.

IOIO_PROT event is one of the NAE events that need to expose register states to the hypervisor. In an IOIO_PROT event, several pieces of information are returned to the hypervisor in GHCB. SW_EXITCODE contains the error code (*i.e.*, 0x7b) of IOIO_PROT events. SW_EXITINFO1 contains the intercepted I/O port (bit 31:16), address length (bit 9:7), operand size (bit 6:4), repeated port access (bit 3), and access type (*i.e.*, IN, OUT, INS, OUTS) (bit 2,0). The SW_EXITINFO2 is used to save the next RIP in non-SEV VM and SEV VM, masked to 0 in SEV-ES and SEV-SNP. For IN instructions, the hypervisor puts the RAX value into the RAX field of GHCB before VMRUN; for OUT instructions, the VC handler places the RAX register value into the RAX field of GHCB before the VMGEXIT.

6.1.2 Cryptographic Side-Channel Attacks

Timing attack. Timing attacks against cryptographic implementations are a subset of side-channel attacks, where the attacker exploits the time difference in the execution of a specific cryptographic function to steal the secret information. Any functions that have secret-dependent execution time variation is vulnerable to timing attacks. However, whether secrets can be stolen in practice depends on many other factors, such as the implementation of the cryptographic function, the hardware supporting the program, the accuracy of the

timing measurements, *etc.*. In 1996, Paul Kocher published the first timing attack on RSA implementation [53]. Boneh and Brumley demonstrated a practical timing attack against SSL-enabled network servers in 2003, where they recovered a server's private key based on the RSA execution time difference [22]. In fact, timing attacks are not only practical against RSA, but to other crypto algorithms, including ElGamal and the Digital Signature Algorithm [71].

Architecture side channel attack. Micro-architecture side channels are side channels that use shared CPU architecture resources to infer a victim program's behaviors. Most micro-architecture side channels exploit timing differences to infer the victim program's behaviors. Some commonly used shared resources in micro-architecture side channels include Branch Target Buffer (BTB), Cache (L1, L2, L3 cache), Translation Look-aside Buffer (TLB) and the CPU internal load/store buffers, *etc.*. Some representative micro-architecture side-channel techniques include Flush+Reload attacks [98], Prime+Probe attack [70], utag attacks [63] and Flush+Flush attacks [36]. Those existing works show that architecture side channels can be exploited and used to break confidentiality in a local or cloud setting.

Constant-time Cryptography. Constant-time cryptography implementations [17] are widely used in mainstream cryptography library to mitigate timing attacks, the design of constant-time functions is used to reduce or eliminate data-dependent timing information. Specifically, Constant-time implementations are making the execution time independent of the secret variables, therefore, do not leak any secret information to timing analysis. To achieve constant execution time, there are three rules to follow. First, the control-flow paths cannot depend on the secret information. Second, the accessed memory addresses can not depend on the secret information. Third, the inputs to variable-time instructions such as division and modulus cannot depend on the secret information. There are a few

tools developed assessing the constant-time implementations, including ImperialViolet [55], dudect [73], ct-verif [2].

6.1.3 Advanced Programmable Interrupt Controller

AMD processors provide an Advanced Programmable Interrupt Controller (APIC) for software to trigger interrupts [6]. Each CPU core is associated with one APIC, and several interrupt resources are supported, including APIC timer, performance monitor counter, and I/O interrupts. In the APIC timer mode, a programmable 32-bit APIC-timer counter can be used by software to generate APIC interrupts. Two modes (periodic and one-shot mode) are supported. In the one-shot mode, the counter can be set to a software-defined initial value and decrease with a clock rate. Once the counter reaches zero, an APIC interrupt is generated on this CPU core. In the period mode, the counter is automatically initialized to the initial value after reaching zero; an interrupt is generated every time the counter reaches zero.

The APIC is used in SGX-Step [24] to single-step the enclave program on Intel SGX [27]. SGX-Step builds a user-space APIC interrupt handler to intercept every APIC timer interrupt. Meanwhile, SGX-Step sets a one-shot APIC timer with a fixed value right before ERESUME. The fixed timer value is configured so that an APIC timer interrupt is generated after a single instruction is executed inside the enclave. These steps are repeated to a single step every instruction inside the enclave. SGX-Step can achieve a single-step ratio of around 98% under a machine-specific fixed counter value. However, as far as we know, no research has studied the APIC timer on the SEV platform to single-step SEV VMs.

6.2 The CIPHERLEAKS Attack

This section explores the side-channel leakage caused by SEV's XEX mode encryption and demonstrates its consequences when applied on the encrypted VMSA page. We particularly construct two attack primitives: execution state inference and plaintext recovery.

6.2.1 The Ciphertext Side Channel

We consider a scenario where the victim VM is a SEV-SNP protected VM hosted by a malicious hypervisor. We assume SEV properly protects the integrity of the encrypted VM memory as well as the VMSA pages. As such, all prior known attacks against SEV and SEV-ES (such as [39, 60, 61, 68, 69, 91]) are not applicable in our setting. The goal of the CIPHERLEAKS attack is to steal secrets from the victim VM. Denial-of-service attacks and speculative execution attacks are out-of-scope.

6.2.1.1 Root Cause Analysis

Because SEV's memory encryption engine uses 128-bit XEX-mode AES encryption, each 16-byte aligned memory blocks in the VMSA is independently encrypted with the same AES key. Since each 16-byte plaintext is first XORed with a physical-address-specific 16-byte value (a.k.a., the output of the tweak function) before encryption, the same plaintext may yield different ciphertext when placed in a different physical address. However, the same 16-byte plaintext is always encrypted into the same ciphertext when placed in the same physical address. Most importantly, SEV (including SEV-ES and SEV-SNP) does not prevent the hypervisor from read accessing the ciphertext of the encrypted memory (which is different from SGX).

This observation forms the foundation of our ciphertext side channel: *By monitoring the changes in the ciphertext of the victim VM, the adversary is able to infer the changes of the corresponding plaintext.* This ciphertext side channel may seem innocuous at first glance, but when applied to certain encrypted memory regions, it may be exploited to infer the execution of the victim VM.

6.2.1.2 CIPHERLEAKS: VMSA Inferences

The CIPHERLEAKS attack is a category of attacks that exploit the ciphertext side channel by making inferences on the ciphertext of the VMSA. We first explain in more details the VMSA structure and then outline an overview of attack.

VMSA structure. Before SEV-ES, the register states were directly saved into VMCB during the VMEXITs without hiding their states from the hypervisor, which gives the hypervisor a chance to inspect the internal states of the VM's execution or change the control flow of software inside the VM []. AMD fixes this unencrypted-register-state vulnerability by encrypting the registers during VMEXITs. In SEV-ES and SEV-SNP, the register states are encrypted and then saved into VMSA during VMEXITs. SEV-ES and SEV-SNP add additional confidentiality and integrity protection of the saved register values in VMSA.

- *Confidentiality.* The VMSA is a 4KB page-aligned memory region specified by the VMSA pointer in VMCB's offset 108h [6]. All register states saved in the VMSA are also encrypted with the VM encryption key K_{vek}.
- *Integrity.* To prevent the hypervisor from tampering VMSA, SEV-ES calculates the hash of the VMSA region before VMEXITs and stores the measurement into a protected memory region. Upon VMRUN, the hardware checks the integrity of the VMSA to prevent any modification of the VMSA data. Instead of performing such integrity checks, SEV-SNP

Table 6.1: Ciphertext of registers collected in the VMSA. If the content at a specific offset is 8 bytes, it means the remaining 8 bytes are reserved.

Offset	Size	Content
150h	16 bytes	CR3 & CR0
170h	16 bytes	RFLAGS & RIP
1D8h	8 bytes	RSP
1F8h	8 bytes	RAX
240h	8 bytes	CR2
308h	8 bytes	RCX
310h	16 bytes	RDX & RBX
320h	8 bytes	RBP
330h	16 bytes	RSI & RDI
340h	16 bytes	R8 & R9
350h	16 bytes	R10 & R11
360h	16 bytes	R12 & R13
370h	16 bytes	R14 & R15

prevents the hypervisor from writing to the guest VM's memory (including VMSA pages) via RMP permission checks.

Overview of CIPHERLEAKS. Our CIPHERLEAKS attack exploits the ciphertext side channel on the encrypted VMSA during VMEXITs. During an AE VMEXIT, all guest register values are stored in the VMSA, which is an encrypted memory page [6]. The encryption of the VMSA page also follows the same rule as other encrypted memory pages. Moreover, as the physical address of the VMSA page is chosen by the hypervisor and remains the same during the guest VM's life cycle, the hypervisor can monitor specific offsets of the VMSA to infer changes of any 16-byte plaintext. Some saved registers and their offset in the VMSA are listed in Table 6.1.

Some 16-byte memory blocks store two 8-byte register values. For instance, CR3 and CR0 are stored at offset 0x150. If either of the two registers changes its value, the corresponding ciphertext will change. Because CR0 does not change very frequently, in most cases, the ciphertext of this block differs because the CR3 value changes, which can

infer a context switch has taken place inside the victim VM. Thus, the ciphertext pair of (CR0, CR3) can be used as identifiers of processes inside the victim VM. For other cases, like the (RBX, RDX) and (R10, R11) pairs, as both registers are subject to frequent changes, it is only possible to learn that the value of one (or both) of the two registers has changed. The adversary may learn which register has changed if she knows the executed binary code between the two VMEXITs.

Some 16-byte memory blocks only store values for a single 8-byte register (*e.g.*, RAX and RCX), and the remaining 8 bytes are reserved. Reserved fields are all 0s, so they never change. Therefore, from Table 6.1, we can see that it is possible to construct one-to-one mappings from the ciphertext to the plaintext for the values of RAX, RCX, RSP, RBP, and CR2.

6.2.2 Execution State Inference

We next describe two attack primitives of CIPHERLEAKS, one in Section 6.2.2 and the other in Section 6.2.3.1. First, we show the use of the ciphertext side channel to infer the execution states of processes inside the guest VM, which helps locate the physical address of targeted functions and infer the executing function of a process.

6.2.2.1 Attack Primitives

To infer the execution states of the encrypted VM, one could follow the steps below:

- ① At time t_0, the hypervisor clears the present bits (P bits) of all memory pages in the victim VM's NPT. The next memory access from the victim VM will trigger a VMEXIT caused by a nested page fault (NPF).
- ② During VMEXITs, the hypervisor reads and records the ciphertext blocks in the victim VM's VMSA, as well as the timestamp and VMEXIT's EXITCODE. Before VMRUN,

Table 6.2: Information revealed from NPF error code.

Bit	Description
Bit 0 (P)	Cleared to 0 if the nested page was non-present.
Bit 1 (RW)	Set to 1 if it was a write access.
Bit 2 (US)	Set to 1 if it was a user access.
Bit 3 (RSV)	Set to 1 if reserved bits were set.
Bit 4 (ID)	Set to 1 if it was a code fetch.
Bit 6 (SS)	Set to 1 if it was a shadow stack access.
Bit 32	Set to 1 if it was a final physical address.
Bit 33	Set to 1 if it was a page table.
Bit 37	Set to 1 if it was a supervisor shadow stack page.

The hypervisor needs to reset the P bit of the faulting page so that the victim VM may continue execution. However, she may choose to clear the P bit again later to trigger more VMEXITs. This step is similar to controlled channel attacks [81, 96].

- ③ The hypervisor collects a sequence of ciphertext blocks and timestamps. By comparing the ciphertext of the CR3 and CR0 fields, the hypervisor may associate each observation to a particular process in the victim VM. Therefore, changes in the ciphertext blocks belonging to the same process can be collected to infer its execution states.

The NPF's error code passed to the hypervisor via VMCB's EXITINFO2 field reveals valuable information for the side-channel analysis. For example, as shown in Figure 6.1b, error code 0x100000014 always means the NPF is caused by an instruction fetch. The NPF error code is specified in Table 6.2.

The ciphertext itself is meaningless, but the fact that it changes matters. We use a vector whose size is the same as the number of registers we monitor to represent value changes in the ciphertext. A value +1 in the vector indicates that the corresponding register has changed since the last NPF. Therefore, a sequence of such vectors can be collected.

With the information described above, the hypervisor is able to profile the applications through a training process.

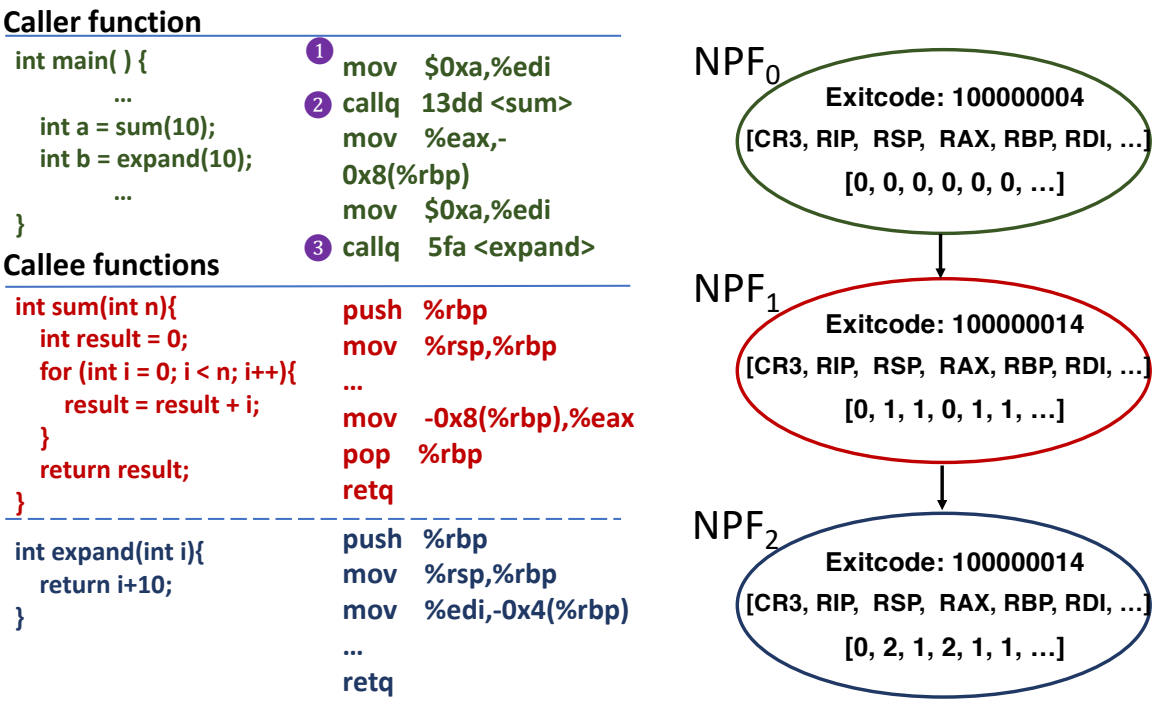
6.2.2.2 Examples

One example of such attack primitives is locating the physical address of targeted functions in the victim. Next, we illustrate such attacks using the example shown in Figure 6.1. We target at two `callq` instructions (❷ and ❸) in the caller function. We assume the hypervisor has some pre-knowledge of the application code running in the guest VM and the hypervisor begins to monitor the application, by clearing the P bits, before the two `call` instructions (*e.g.*, before ❶). In handling each NPFs, the hypervisor collects the ciphertext of those saved registers listed in Table 6.1 as well as the NPF's error code.

The hypervisor then collects a sequence of ciphertext blocks as shown in Figure 6.1b. The `callq` instruction at ❷ touches a new instruction page that contains the code of `sum()`. Therefore it triggers an NPF. Compared to the previous snapshot, the changes of the ciphertext of RIP, RSP, RBP, and RDI are observed; the ciphertext of CR3 and RAX remains unchanged. When `sum()` returns, the return value is stored in RAX. The ciphertext changes of the RAX register will be observed in the next NPF (at ❸), where RIP will also change. In this way, the hypervisor can locate the physical address of the functions and trace the control flow of the target application. In particular, NPF_1 reveals the physical address of function `sum()`, NPF_2 reveals the physical address of `expand()`.

6.2.3 Plaintext Recovery

The ciphertext side channel can also be exploited to recover the plaintext from some of the ciphertext blocks. To recover plaintext from the ciphertext, the adversary first needs to build a dictionary of plaintext-ciphertext pairs for the targeted registers, and then make use of the dictionary to recover the plaintext value of the registers of interest during the execution of a sensitive application.



(a) C source code with assembly code.

(b) Ciphertext blocks.

Figure 6.1: Example about the ciphertext changes in NPFs.

6.2.3.1 Attack Primitive

During some NAE events, the guest kernel may exchange register states with the hypervisor through GHCB. Thus, the plaintext value of specific registers can be learned when these register states are stored in the GHCB. The hypervisor can thus collect plaintext-ciphertext pairs for those registers. Because different registers have different offset in the VMSA and different physical addresses, we need to build the dictionary of plaintext-ciphertext pairs for each register separately.

There are two ways to collect such pairs, depending on who stores the register values to GHCB. First, for those NAE events where the hypervisor returns emulated registers to the guest VM, the hypervisor may clear the P bit of the instruction page that triggers

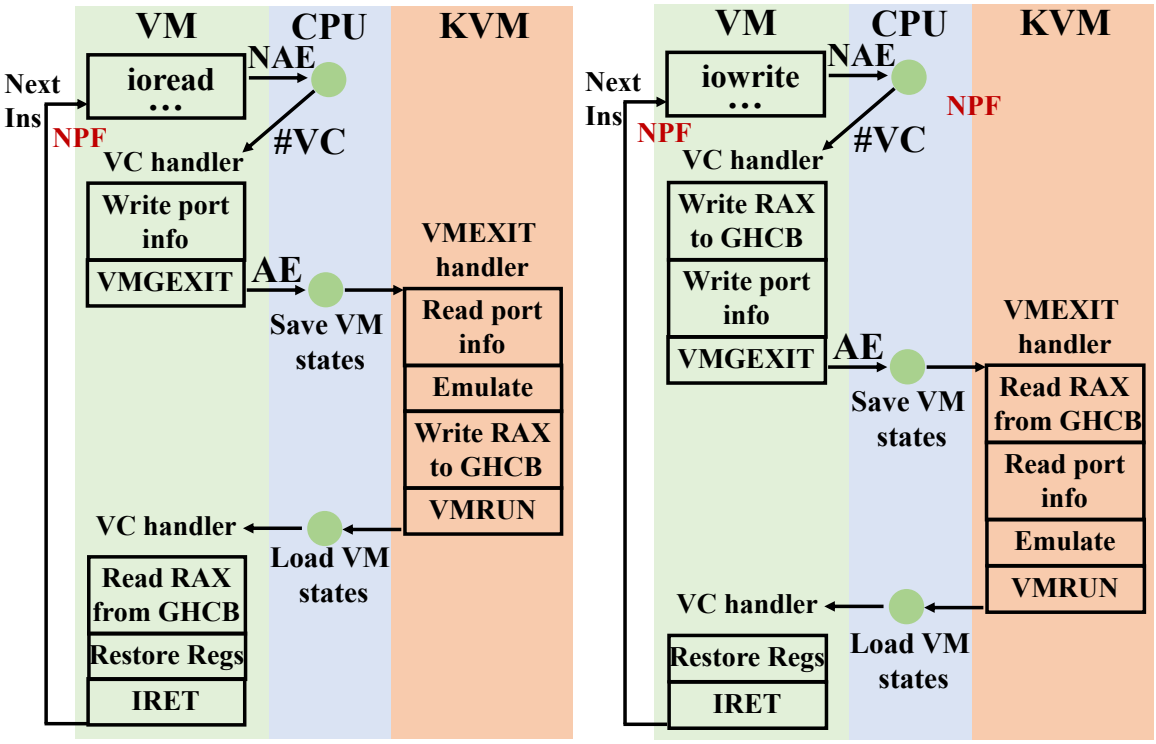
the NAE events before VMRUN. Thus, after the VC handler use `IRET` to return to the original instruction page, an NPF will occur, and the hypervisor can obtain the ciphertext of corresponding registers while handling this NPF. Figure 6.2a shows an example about collecting plaintext-ciphertext pairs of RAX from IOIO_PROT events (`ioread`). The hypervisor records the plaintext of RAX when emulating the VMEXIT and obtains the ciphertext of RAX when handling the NPF caused by `IRET`.

Second, for those NAE events where the VM exposes registers to the hypervisor, the hypervisor may periodically clear the P bit of the VC handler code and record the ciphertext of all registers in VMSA whenever there is an NPF triggered by the VC handler code. At the next NAE, the plaintext of some registers will be written to the GHCB, and their corresponding ciphertext can be found from the last VC handler triggered NPF. Figure 6.2b shows an example about collecting plaintext-ciphertext pairs of RAX from IOIO_PROT events (`iowrite`). The hypervisor obtains the ciphertext of RAX either when handling the VC-exception-triggered NPF after the NAE event or when handling the NPF caused by `IRET` and learns the plaintext of RAX when handling the VMEXIT.

6.2.3.2 Examples

The adversary could use the NAE VMEXITs to collect a dictionary of plaintext-ciphertext pairs for certain registers stored in VMSA. Here we present a method that leverages the IOIO_PROT (error code = 0x7b) NAE VMEXIT events to collect the ciphertext of the RAX register when its plaintext values are 0 to 127.

Building the dictionary of plaintext-ciphertext pairs. During the PEI phase, the guest VM needs to access the memory region that stores the information about the Nonvolatile BIOS settings (CMOS) and the Real-Time Clock (RTC) through IO ports 0x70 and 0x71.



(a) `ioread` event.

(b) `iowrite` event

Figure 6.2: Workflow of how VC handler handles IOIO_PROT events.

The OVMF code ensures the correctness of the CMOS/RTC by calling a function named `DebugDumpCmos` when loading the `PlatformPei` PEI Module (PEIM) during the initialization of the guest VM. `DebugDumpCmos` checks the CMOS/RTC by writing the offset of CMOS/RTC to port 0x70 and then reading one byte of data from port 0x71. `DebugDumpCmos` enumerates offset 0x00-0x7f (*i.e.*, 0-127) during the PEI phase to access the CMOS/RTC information.

In both SEV-ES and SEV-SNP, every `iowrite` and `ioread` in IOIO_PROT are first trapped and handled by the VC handler. The VC handler and the hypervisor then cooperate to emulate `iowrite` and `ioread` as shown in Figure 6.2. For `iowrite`, the VC handler copies the RAX value to GHCB before calling `VMGEXIT`. For `ioread`, the VC handler copies

Table 6.3: Number of NAE events observed during boot period and registers state range maybe exposed. Num: the number of NAE event being observed. *: state to hypervisor. **: state from hypervisor, N/A: not observed. -: this register is not supposed to be used during this NAE event. Range R1: numbers of different exposed register states lying in [0,1], Range R2: [0,15], Range R3: [0,127], Range R4: [0,2^{64}-1].

NAE Event	Num	RAX				RCX				RDX			
		R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
DR7 Read*	0	N/A	N/A	N/A	N/A	-	-	-	-	-	-	-	-
DR7 Write*	1	0	0	0	1	-	-	-	-	-	-	-	-
RDTSC*	0	N/A	N/A	N/A	N/A	-	-	-	-	N/A	N/A	N/A	N/A
RDPMC*	0	-	-	-	-	N/A	N/A	N/A	N/A	-	-	-	-
RDPMC**	0	N/A	N/A	N/A	N/A	-	-	-	-	N/A	N/A	N/A	N/A
CPUID*	35328	2	6	6	276	2	11	18	1467	-	-	-	-
CPUID**	35328	1	5	6	18	1	2	3	17	2	3	4	10
IOIO_PROT*	260648	2	16	128	8717	-	-	-	-	-	-	-	-
IOIO_PROT**	246527	2	15	82	9033	-	-	-	-	-	-	-	-
RDMSR*	1261	-	-	-	-	0	0	1	104	-	-	-	-
RDMSR**	1261	2	4	4	51	-	-	-	-	1	1	2	6
WRMSR*	12532	1	4	6	10363	0	0	1	71	1	1	2	8
RDTSCP**	0	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

the RAX state from GHCB to RAX register after `VMGEXIT`. In the `iowrite` cases, the RAX state after the VC handler finishing handling an iowrite exception and before returning to the sequential instruction, should be the same as the RAX state passed to the hypervisor in the `VMGEXIT`.

In our case of `DebugDumpCmos` in `PlatformPei` PEIM, the hypervisor can observe 128 IOIO_PROT events with SW_EXITINFO1 being 0x700210 (indicating that the guest VM is accessing CMOS/RTC information) and increasing RAX values from 0x00 to 0x7f. The hypervisor can also trap the sequential instruction by clearing the P bit of the physical address of the `PlatformPei` PEIM's *EntryPoint*, which will be accessed after the guest VM exiting the VC handler. The guest physical address of *EntryPoint* is always 0x83a000 in our setting. Note that the hypervisor can also easily locate the physical address of the

`PlatformPei` PEIM because the plaintext of the OVMF file is known by both the guest VM owner and the hypervisor [7] for in-place encryption during the remote attestation.

Each IOIO_PROT event in `DebugDumpCmos` helps the hypervisor record the ciphertext of a known RAX plaintext value in VMSA when handling the NPF caused by returns to the `PlatformPei` PEIM. After the `DebugDumpCmos`, the hypervisor can build a dictionary with 128 plaintext-ciphertext pairs in total, where the plaintext are from 0x00 to 0x7F. Some other IOIO_PROT events with the same SW_EXITINFO1 can also occur during the execution of `DebugDumpCmos`. The hypervisor can distinguish those events by looking at the ciphertext of RFLAG/RIP field in VMSA since all target `iowrites` inside `DebugDumpCmos` have the same RFLAG/RIP value.

6.2.3.3 Other Plaintext-ciphertext Pairs

In this section, we show other plaintext-ciphertext pairs the adversary may collect during the boot period of a SEV-enabled VM. We also analyze plaintext recovery under different OVMF versions and different build configurations.

All data shown in this section were collected on a workstation with 8-Core AMD EPYC 7251 Processor. The OVMF version used to boot the SEV-ES-enabled VMs may vary according to different settings that we will illustrate later. The victim VMs were configured as SEV-ES-enabled VMs with one virtual CPU, 4 GB DRAM, and 30 GB disk storage. The host and guest OS kernel were forked from branch sev-es-v3, and the QEMU version was QEMU sev-es-v12. All code is directly downloaded from AMD's Github repository [9] (commit:`96f2b75aaa9801646b410568d12b928cc9f06e0c`, Nov, 25th, 2020). We only performed the attacks on SEV-ES machines, as SEV-SNP machines were not available to us at the time of writing. But SEV-SNP is equally vulnerable (see Section 5.6).

Plaintext Range. To show the potential plaintext range the hypervisor can collect, we monitored all NAE events which have register state interactions with the hypervisor during the boot period of a SEV-ES-enabled VM. The OVMF version used was downloaded from branch sev-es-v27 with the default setting. As shown in Table 6.3, the collected register states are divided into 4 intervals. Range 1 (R1) is field [0,1] with only two numbers and is the most important interval since a return of true or false is very common in function implementation. Most observed NAE events can help the hypervisor to collect both two values in R1 while frequent IOIO_PROT (260648 for IO out and 246527 for IO in) events during the boot period can help the hypervisor to fill Range 2 (R2) which is [0,15] and Range 3 (R3) which is [0,127]. Range 4 (R4) contains all 2^{64} for an 8-byte register. Some NAE events are not observed during the boot period like `RDPMC` and `RDTSC`. However, these NAE events are still considered exploitable as long as some programs use these instructions during VM's lifetime. In the table, we separate RBX and RDX to present different register values the hypervisor can observe during the boot period. However, the adversary is only able to observe the ciphertext of the (RBX, RDX) pair, as these two registers are in an the same aligned 16-byte encryption block.

Different Versions. We have tested three latest (as of Nov., 25th, 2020) OVMF git branches provided by AMD [9] for SEV-ES ("sev-es-v27"[7]) and SEV-SNP ("sev-es-v21+snp"[8]) as well as the official OVMF repository used by SEV ("https://github.com/tianocore/edk2.git"[9]). All these three versions adopt the same CMOS/RTC design flow we mentioned in this section under the default configuration provided by AMD [9], and the hypervisor is able to collect all the 7-bits (plaintext from 0 to 0x7F) plaintext-ciphertext pairs in all these three versions.

[7] commit:`834f296d3e1864b676fac9db53bc7dbb83c6eee7`

[8] commit:`e7bf4dfeaba60089f427af518936f29db79dd159`

[9] commit:`21f984cedec1c613218480bc3eb5e92349a7a812`

Different Settings. We have also tested OVMF debug configuration options. The default debug configuration is to write debug messages to IO port 0x402. OVMF also supports original debug behavior where the debug messages are written to the emulated serial port if the `DEBUG_ON_SERIAL_PORT` option is set. AMD adopts the `DEBUG_ON_SERIAL_PORT` option according to their Github repository [9]. In both these two settings, the hypervisor is able to collect all the 7-bits plaintext-ciphertext pairs by monitoring CMOS/RTC activities in I/O PORT 0x70. The `DebugDumpCmos` can be disabled if the developer chooses to ignore all debug information by setting the `-b RELEASE` option. However, the hypervisor can still collect 19 out of the 7-bits plaintext-ciphertext pairs (with 2 numbers lying in R1, 13 numbers in R2, and 19 numbers in R3) by monitoring CMOS/RTC activities in I/O PORT 0x70. When targets at all IOIO_PROT OUT events, the hypervisor shows the potential ability to collect 115 out of the 7bits plaintext-ciphertext pairs (with 2 numbers lying in R1, 16 numbers in R2, and 115 numbers in R3), even disabling all debug activities.

6.3 Case Studies

In this section, we present two case studies to illustrate the CIPHERLEAKS attack. In the first attack, we show that the constant-time RSA implementation in OpenSSL can be broken with known ciphertext for the plaintext values of 0 to 31. In the second attack, we show that the constant-time ECDSA signature can be compromised with known ciphertext of the plaintext values of 0 and 1.

6.3.1 Breaking Constant-Time RSA

RSA is asymmetric cryptography, which is widely used in various crypto systems. In the RSA algorithm, the plaintext message `m` can be recovered from the ciphertext `c` via $m = c^d$

mod n, where d is the private key and n is the modulus of the RSA public key system. As such, we show how the CIPHERLEAKS attack steals the private key d.

Targeted RSA implementation. Our demonstrated attack targets at the modular exponentiation used in RSA operations from the latest OpenSSL implementation (as of Nov, 4th, 2020)[10] . OpenSSL implements the modular exponentiation using a fixed-length sliding window method in function `BN_mod_exp_mont_consttime()`. We target at a *while* loop inside this function, which iteratively calculates the exponentiation in a 5-bit windows. The *while* loop is shown in Listing 6.1. For a 2048-bit private key, the *while* loop has about 2048/5 = 410 iterations. In each iteration, `bn_get_bits5` is called to retrieve the 5-bit of the private key `d`.

```
/*
 * Scan the exponent one window at a time starting from the most
   significant bits.
 */
while (bits > 0) {
    bn_power5(tmp.d, tmp.d, powerbuf, np, n0, top,
              bn_get_bits5(p->d, bits -= 5));
}
```

Listing 6.1: Code snippet of `BN_mod_exp_mont_consttime`.

The attacker can steal the 2048-bit private key `d` in the following steps:

① **Infer the physical address of the target function.** The attacker first uses the method introduced in Section 6.2.2 to obtain the physical address of the target function. We use gPA_{t0} and gPA_{t1} to denote the guest physical addresses of the target functions `bn_power5` and `bn_get_bits5`, respectively.

② **Monitor NPFs.** The attacker clears the P bit of the two targeted physical pages. Once a NPF of gPA_{t0} is intercepted, she clears the P bit of gPA_{t1}; when a NPF of gPA_{t1} is intercepted,

[10]Github commit: `8016faf156287d9ef69cb7b6a0012ae0af631ce6`

she clears the P bit of gPA_{t0}. For a 2048-bit RSA encryption, 410 iterations can be observed, the attacker will observe 820 NPFs of gPA_{t0} and gPA_{t1} in total.

③ **Extract the private key d.** As shown in Listing 6.2, `bn_get_bits5` obtains 5 bits of d in each iteration, stores the value in RAX, and returns. Since the hypervisor clears the P bit of gPA_{t0}, returns to `bn_power5` will trigger a NPF of gPA_{t0}. When the hypervisor handles this NPF, it reads and records the ciphertext of RAX in the VMSA. The RAX now stores 5 bits of the private key d, and its value range is 0 to 31. The hypervisor can infer the plaintext by searching the plaintext-ciphertext pairs collected during the boot period as described in Section 6.2.3.2. The hypervisor can recover the whole 2048-bit private key d after a total of 410 iterations.

```
1 .globl  bn_get_bits5
2     ......
3     cmova %r11,%r10
4     cmova %eax,%ecx
5     movzw (%r10,$num,2),%eax
6     shrl  %cl,%eax
7     and \$31,%eax
8     ret
9     ......
```

Listing 6.2: Code segment of `bn_get_bits5()`.

6.3.2 Breaking Constant-time ECDSA

Elliptic Curve Digital Signature Algorithm ECDSA) is a cryptographical digital signature based on the elliptic-curve cryptography (ECC). ECDSA follows the steps below to generate a signature:

1. Randomly generate a 256-bit nonce k.
2. Calculate $r = (k \times G)_x \mod n$
3. Calculate $s = k^{-1}(h(m) + rd_a) \mod n$

where G is a base point of prime order on the curve, n is the multiplicative order of the point G, d_a is the private key, $h(m)$ is the hash of the message m, and (r, s) form the signature. With a known nonce k, the private key d_a can be calculated directly:

$$d_a = r^{-1} \times ((ks) - h(m)) \mod n$$

As such, a side-channel attack against ECDSA aims to steal the nonce k. The secret private key can be inferred thereafter.

Targeted ECDSA implementation. Our demonstrated attack targets the `secp256k1` curve, which is also used in Bitcoin wallets. In the latest OpenSSL's implementation (as of Nov, 4th, 2020) , when `ECDSA_do_sign` is called to generate a signature, `ecdsa_sign_setup` is first called to generate a random 256-bit nonce k per NIST SP 800-90A standard. To do so, `EC_POINT_mul`, `ec_wNAF_mul`, and then `ec_scalar_mul_ladder` are called to compute r, which is the x-coordinate of nonce k. `ec_scalar_mul_ladder` is used regardless of the value of the `BN_FLG_CONSTTIME` flag.

As shown in Listing 7.1, the core component of `ec_scalar_mul_ladder` uses conditional swaps (*a.k.a.*, `EC_POINT_CSWAP`) to compute point multiplication without branches. Specifically, in each iteration, `BN_is_bit_set(k, i)` is called to get the i^{th} bit of the nonce `k`. The conditional swaps are determined by `kbit`, which is the XOR result of the i^{th} bit of the nonce `k` and `pbit`.

```
for (i = cardinality_bits - 1; i >= 0; i--) {
    kbit = BN_is_bit_set(k, i) ^ pbit;
    EC_POINT_CSWAP(kbit,r,s,group_top,Z_is_one);
// Perform a single step of the Montgomery ladder
    if (!ec_point_ladder_step(group, r, s, p, ctx)){
        ERR_raise(ERR_LIB_EC,
        EC_R_LADDER_STEP_FAILURE);
        goto err;
    }
// pbit logic merges this cswap with that of the next iteration
    pbit ^= kbit;
```

```
}
```

Listing 6.3: Code snippet of `ec_scalar_mul_ladder()`.

The attacker can steal the nonce `k` in the following steps:

① **Infer the functions' physical addresses.** The attacker first obtains the guest physical addresses of the target functions `ec_scalar_mul_ladder` gPA_{t0} and `BN_is_bit_set` gPA_{t1} using the execution inference method we introduced.

② **Monitor NPFs.** The attacker clears the P bit of the two targeted physical pages. Once a NPF of gPA_{t0} is intercepted, she clears the P bit of gPA_{t1}; when a NPF of gPA_{t1} is intercepted, she clears the P bit of gPA_{t0}. In this way, the control flow internal to the `ec_scalar_mul_ladder` function can be learned by the attacker.

③ **Learn the value of `k`.** In the 256-iteration *while* loop, the attacker will observes 256*5 = 1280 NPFs of gPA_{t0} and 1280 NPFs of gPA_{t1}. In each iteration of the *while* loop, the first NPFs of gPA_{t0} is triggered when `BN_is_bit_set` returns. As shown in Listing 6.4, the i^{th} bit of the nonce `k` is returned in RAX. Thus, the i^{th} bit of the nonce `k` is stored in the RAX field of the VMSA for the first NPFs of gPA_{t0} in each iteration. The attacker then compares the ciphertext of the RAX field to recover the nonce `k`.

```
000f8e20 <BN_is_bit_set>:
             ......
f8e38:  48 8b 04 d0    mov    (%rax,%rdx,8),%rax
f8e3c:  48 d3 e8       shr    %cl,%rax
f8e3f:  83 e0 01       and    $0x1,%eax
f8e42:  f3 c3          repz retq
             ......
```

Listing 6.4: Assembly code snippet of `BN_is_bit_set()`.

6.3.3 Evaluation

All end-to-end attacks shown in this section were evaluated on a workstation with 8-Core AMD EPYC 7251 Processor. The victim VM was configured as SEV-ES-enabled VMs with

one virtual CPU, 4 GB DRAM, and 30 GB disk storage. The versions of the guest and host OS, QEMU, and OVMF are the same as described in Section 6.2.3.3. The latest OpenSSL from Github was used in the evaluation (commit:`8016faf156287d9ef69cb7b6a0012ae0af631ce6`, Nov, 4th, 2020). These attacks can also be applied to VMs with multiple vCPUs as well, but the adversary needs to collect ciphertext-plaintext dictionaries for each vCPU independently, since each vCPU has its own VMSA.

To locate the physical address of the target function, the attacker must train the pattern of ciphertext changes in a training VM (a different VM from the victim VM). In the training VM, the attacker first repeats the RSA encryption and the ECDSA signing several times by calling APIs from the OpenSSL library (with the same version as the targeted OpenSSL library in the victim VM). The attacker also collects the NPF sequence, the corresponding VMSA ciphertext changes (see Section 6.2.2), as well as the ground truth (guest physical address) for the target functions. In our experiments, the pattern of ciphertext changes is very stable, especially for a function call without many branches (*e.g.*, ECDSA_do_sign() for ECDSA). As such, simple string comparison is sufficient for pattern matching and no sophisticated machine learning techniques are required.

In the attack phase, the victim VM performs an RSA encryption or an ECDSA signature using the OpenSSL library, which can be triggered by the attacker remotely but it is not a necessary condition for a successful attack. As the attacker does not know the start time of the targeted program, she must consider every newly observed CR3 ciphertext as the beginning of the targeted crypto code. It clears all P bits and starts monitoring the pattern of ciphertext changes. If the expected ciphertext change pattern is observed, the attacker can continue to steal the secret from the victim VM.

In both of the two cases we presented, we repeated the experiment 10 times and each time the attacker was able to identify the trained ciphertext pattern and recover the private key d and the secret nonce `k` with 100% accuracy. We measured the time needed to steal the 2048-bit private key `d` and the secret nonce `k` 10 times after the ciphertext change pattern is identified. The average time needed to obtain the private key `d` is 0.40490 seconds with a standard deviation of 0.08920 seconds. The average time needed to steal the secret nonce `k` is 0.10226 seconds with a standard deviation of 0.00330 seconds.

6.4 Countermeasures

In this section, we first discuss several potential software-level countermeasures for the CIPHERLEAKS attack. We then show the CIPHERLEAKS attack can still work by exploiting the Advanced Programmable Interrupt Controller (APIC) to collect the function's internal state. Thus, none of that software may work properly. We also discuss hardware-level countermeasures in Section 6.4.3.

6.4.1 Software Mitigation

Solutions to the ciphertext side channel can be categorized into two kinds: preventing the collection of the plaintext-ciphertext dictionary and preventing exploitation by modifying targeted functions.

Preventing dictionary collection. One potential solution is to remove unnecessary IOIO_PROT events. However, other NAE event may still serve the same purposes as IOIO_PROT. More importantly, as we have shown in Section 6.3.2, the hypervisor can steal the nonce k with only two plaintext-ciphertext pairs. Complete removal of all such leak sources is required to make the solution effective, almost impossible in SEV's current design.

Preventing exploitation. To fix the target functions, changes to the whole software stack may be necessary. We list three potential solutions below, but unfortunately, these approaches can be bypassed using the method we outline in Section 6.4.2.

- **Masking the return value in RAX.** If the return value only needs a few bits to represent, compilers can introduce randomness into the higher bits of the return value. For example, if the return is 1, then a random number can be added to mask the RAX, *e.g.*, by returning RAX = 0x183af6b800000001, where the higher 4-byte are generated randomly. The caller of the function can ignore the higher bits. In this way, the ciphertext of RAX will be new and thus unknown to the adversary.
- **Passing return values through memory or other registers.** The return value can be passed to the caller via stack. As the physical address of the stack frame is hard to predict and collect beforehand, attacks can be prevented. Similarly, the software can also write the return value to other registers (*e.g.*, R10), which can avoid using the RAX register.
- **Using inline functions or keep the callee code on the same page.** If the code of the caller and the callee are on the same page, for instance, by using inline functions, no NPFs will be triggered during function return.

These three potential solutions require significant rewriting of sensitive functions, which may require compiler-assisted tools to perform. However, the success of all these solutions relies on the assumption that the hypervisor cannot infer the internal states of a function call, which, as we will show in Section 6.4.2 shortly, is not true.

6.4.2 Function's Internal States Intercept

We present an APIC-based method to allow the hypervisor to single-step the functions in order to intercept the function's internal states. Therefore, the adversary can learn the

internal states of a targeted function. Our method, though conceptually similar to SGX-Step [24], requires integrating the APIC handling code into the VMEXIT handler of KVM. Moreover, unlike SGX-Step that uses a static APIC interval to interrupt the controller, we need to select APIC intervals as the execution time of VMRUN is not constant. More specifically, the following steps are taken to interrupt VMRUN:

① **Infer the functions' physical addresses.** The attacker first obtains the guest's physical addresses of the target function, namely gPA_t, using the execution state inference method we introduced.

② **Dynamically determine APIC timer intervals.** The attacker follows a "0 steps is better than several steps" principles to single step or intercept a small advancement of the execution of the target function. Because the time used for VMRUN instruction is not fixed, the hypervisor always starts with a small APIC interval to single step into the guest VM as much as possible. The hypervisor then checks the VMSA field to see if the ciphertext in VMSA has changed; if so, it means that one or several registers' value have changed and the guest VM executes one or several instructions before interrupted by APIC. The algorithm to choose the proper APIC time interval is specified in Algorithm 2.

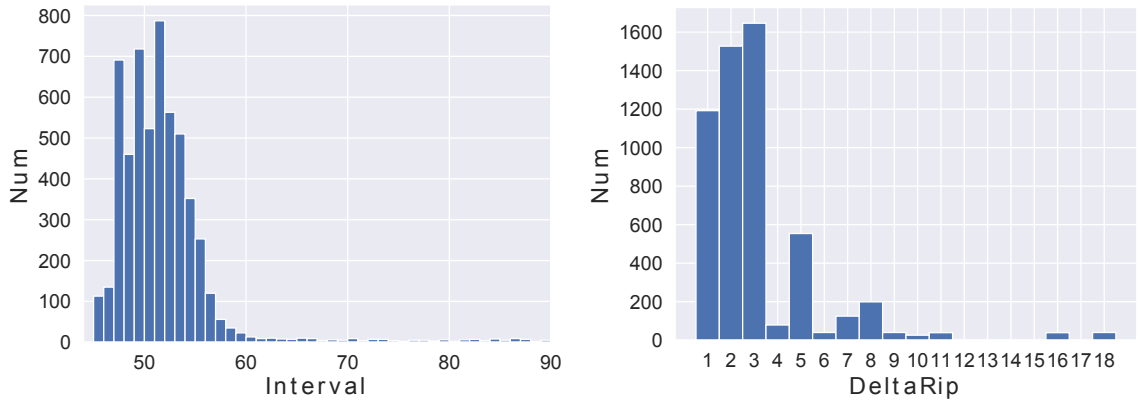
Algorithm 2 Dynamic Timer Interval Prediction

```
int apic_time_interval; //APIC interrupts the VM after the interval
int roll_back ; //roll back to a small interval after any movement
apic_time_interval = 20
roll_back = 10; // initialize the setting, may vary in different CPU
while true do
    apic_timer_oneshot(apic_time_interval)                    __svm_sev_es_vcpu_run(svm->vmcb_pa)
    svm_handle_exit(vcpu, physical interrupt VMEXIT)   if not observe VMSA changes then
        apic_time_interval ++
    else
        apic_time_interval = apic_time_interval - roll_back
    end
end
```

③ **Collect the target function's internal states.** The hypervisor can collect the internal states of the target function after a WBINVD instruction which is used to flush VMSA's cache back to the memory. With a known binary, the hypervisor may also determine the number of the instructions that have been executed by comparing the ciphertext blocks changes with the assembly code.

Evaluation. To evaluate the effectiveness of single-stepping the guest VM's execution, we perform experiments on a workstation with 8-Core AMD EPYC 7251 Processor. The victim VM was configured as SEV-ES-enabled VMs with two virtual CPUs, 4 GB DRAM, and 30 GB disk storage. The versions of the guest and host OS, QEMU, and OVMF are the same as described in Section 6.2.3.3. Unlike the previous settings, we enable SEV-ES's debug option in the guest policy, which allows the hypervisor to use `SEV_CMD_DBG_DECRYPT` command to decrypt the guest VM's VMSA. This configuration is only to collect ground truth of the experiments, which will not influence the guest VM's execution and is not a required step in practical attacks.

To make the experiments representative, we randomly select the starting point during the VM's execution to initiate our tests. In each test, we follow Algorithm 2 to collect 100 trials. Each trial is collected only when the hypervisor observes changes in the register's ciphertext in the VMSA. Meanwhile, we collected ground truth by using the `SEV_CMD_DBG_DECRYPT` command from the hypervisor to decrypt the RIP filed in VMSA. We use Δ to represent the number of bytes that the RIP has advanced between two consecutive VMEXITs. Note that the `SEV_CMD_DBG_DECRYPT` command will not affect the execution of the guest VM. We repeat the test 60 times. In total, 6000 trials are collected. Among the 6000 trials, 454 lead to Δ greater than 20 because of a `jmp` instruction (thus can be filtered out). For the remaining 5546 trials, the APIC-timer intervals used to trigger APIC interrupts range from



(a) Interval when VMSA changes.

(b) Δ when VMSA changes.

Figure 6.3: Performance of stepping VM execution using APIC.

40 to 90 (with a divide value of 2, this translates from 80 to 180 CPU cycles). The distribute is shown in Figure 6.3a. These results suggest that the runtime of the VMRUN instruction is not constant (on SEV-ES VM), which may be caused by the presence of VMCB cache states and the non-constant time VMSA integrity checks. Even though VMRUN is not constant-time, as shown in Figure 6.3b, 78.7% trials lead to Δ smaller than 3 bytes. 90.1% trials lead to Δ smaller than 5 bytes. Note that a typical x86 instruction has 2 to 4 bytes [41]. These results show that the APIC-based method can successfully interrupt the execution of the guest VM with very small steps.

6.4.3 Hardware Countermeasures

The root cause of the ciphertext side channel is the mode of encryption adopted in the memory encryption. AMD uses the XEX encryption mode in all SEV versions (*e.g.*, SEV, SEV-ES, and SEV-SNP) and all CPU generation (*e.g.*, Zen, Zen 2, and Zen 3). This results from a well-known dilemma in the design of memory encryption: On one hand, if the ciphertext of each 16 blocks is chained together (like in the CBC mode encryption), the

static mapping between ciphertext and plaintext can be broken. However, changing one bit in the plaintext will lead to changes in a large number of ciphertext blocks. On the other hand, if freshness is introduced to each block (like the CTR mode encryption used in Intel SGX), a large amount of memory needs to be reserved for storing the counter values. However, this idea may be applied to only selected memory regions, such as VMSA. In this way, the CIPHERLEAKS attack against VMSA can be prevented. To our knowledge, the hardware patch that will be integrated in SEV-SNP takes a similar idea for protecting VMSA. However, the ciphertext side channel still exists in other memory regions.

Alternatively, a plausible hardware solution is to prevent the hypervisor's read accesses to the guest VM's memory. This idea could be implemented with the RMP table (see Section 5.6), by restricting the read access from the hypervisor on guest pages. However, this feature is not yet available in SEV-SNP.

6.5 Applicability to SEV-SNP

To mitigate memory integrity attacks against SEV and SEV-ES [61, 69, 91, 93], AMD introduced another extension of SEV, named SEV Secure Nested Paging (SEV-SNP) [50]. AMD released the whitepaper describing in January, 2020 [8] and a hardware API document in August, 2020 [10]. Nevertheless, commercial processors supporting SEV-SNP have not been released yet. According to the technical details revealed in SEV-SNP's whitepaper, all prior attacks (at the time of August, 2020) can be mitigated by SEV-SNP.

In this section, we discuss some of the new features introduced by SEV-SNP and discuss CIPHERLEAKS's applicability on SEV-SNP.

6.5.1 Overview of SEV-SNP

SEV-SNP protects guest VM's memory integrity by introducing a new structure called Reverse Map Table (RMP). Each RMP entry is indexed by the system page frame numbers; it contains the page states (*e.g.*, page's ownership, guest-valid, guest-invalid, and guest physical address) of this system page frame. The SEV-SNP VM must interact with the hypervisor to validate each RMP entry. Specifically, the guest VM needs to issue a new instruction `PVALIDATE`, a new instruction for guest VMs, to validate a guest physical address before the first access to that guest physical address. Any memory access to an invalid guest physical address will result in an NPF. More importantly, once a guest page is validated, the hypervisor cannot modify the RMP entry. Therefore, the guest VM itself can guarantee that its memory page is only validated once, and a one-to-one mapping between the guest physical address and system physical address mapping can be maintained.

As shown in Figure 6.4, RMP limits the hypervisor's capabilities of managing NPT. The RMP check is performed before the NPT walk is finished. Without RMP check, the hypervisor can easily remap guest physical address (*gPA*) to an arbitrary memory page by manipulating the page table entry in the NPT. With RMP check, if the hypervisor remaps the guest physical address to a memory page not belonging to the current guest VM or a memory page mapped to the current guest VM's other guest physical address, an invalid NPF or a mismatch NPF will be triggered, which can prevent attacks that require modification of the NPT [39, 68, 69].

Another protection enabled by RMP is that the ownership included in the RMP entry restricts the hypervisor's write permission towards the guest VM's private memory, which can prevent attacks that require directly modifying the ciphertext [29, 61, 93]. More details about existing attacks and how RMP can mitigate these attacks are introduced in Section 2.2.

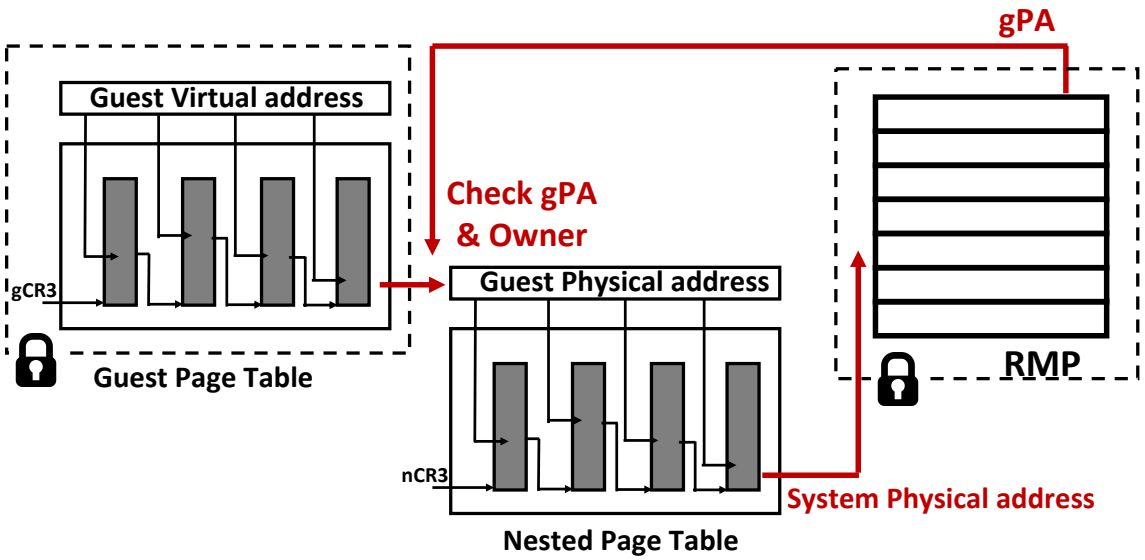


Figure 6.4: The RMP Check in AMD-SNP.

6.5.2 The CIPHERLEAKS attack on SEV-SNP

There are two key requirements of the CIPHERLEAKS attack:

- **Mapping of plaintext-ciphertext pairs of the same address does not change.** When applying the CIPHERLEAKS attack on SEV-SNP, the memory encryption mode in SEV-SNP needs to preserve the mapping between the plaintext and the ciphertext throughout the lifetime of the VM. According to [6], SEV-SNP still adopts the XEX mode of encryption, which satisfies this requirement.
- **The hypervisor must have read access to the ciphertext.** When applying the CIPHERLEAKS attack on SEV-SNP, the adversary needs to have read access to the ciphertext of guest VM's memory. According to [8], even though RMP limits the hypervisor's write access towards VM's private memory, the hypervisor still has read access to the guest VM's memory, including the VMSA area.

AMD has confirmed that SEV-SNP is also vulnerable to the CIPHERLEAKS attack. A CVE number will be assigned the discovered vulnerability for SEV-SNP and a hardware patch will be available to protect the VMSA during VMEXITs.

6.6 Summary

In this chapter, we describes the ciphertext side channel on SEV (including SEV-ES and SEV-SNP) processors. The root causes of the side channel are two-fold: First, SEV uses XEX mode of encryption with a tweak function of the physical addresses, so that the one-to-one mapping between the ciphertext and plaintext of the same address is preserved. Second, the VM memory is readable by the hypervisor, allowing it to monitor the changes of the ciphertext blocks. The chapter demonstrates the CIPHERLEAKS attack that exploits the ciphertext side-channel vulnerability to completely break the constant-time cryptography of OpenSSL when executed in SEV-ES VMs.

Chapter 7: A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP

In this chapter, we perform a comprehensive study on the ciphertext side channels. Our work suggests that while the CipherLeaks attack targets only the VMSA page, a generic ciphertext side-channel attack may exploit the ciphertext leakage from any memory pages, including those for kernel data structures, stacks and heaps. As such, AMD's existing countermeasures to the CipherLeaks attack, a firmware patch that introduces randomness into the ciphertext of the VMSA page, is clearly insufficient. The root cause of the leakage in AMD SEV's memory encryption—the use of a stateless yet unauthenticated encryption mode and the unrestricted read accesses to the ciphertext of the encrypted memory—remains unfixed. Given the challenges faced by AMD to eradicate the vulnerability from the hardware design, we propose a set of software countermeasures to the ciphertext side channels, including patches to the OS kernel and cryptographic libraries. We are working closely with AMD to merge these changes into affected open-source projects.

Responsible disclosure. We disclosed the generic ciphertext side-channel attacks on kernel data structures, heaps, and stacks to the AMD SEV team in August 2021. Henceforth, we provided more supplementary materials via email communications. AMD has acknowledged the vulnerability and had several discussions with us about potential countermeasures and stated interest in a kernel level fix. While hardware countermeasures might not be feasible

in the near future for both performance and design concerns, AMD assisted us with the development of the software countermeasures, including both kernel patches (Section 7.5) and helping us get connected to other projects like OpenSSL.

We also disclosed the vulnerability on the code level to the communities of cryptography libraries (including OpenSSL, WolfSSl, GnuTLS, OpenSSH and libgcrypt). At the time of writing, we had received feedback from both OpenSSL and WolfSSL. They both acknowledged the concerns and recognized the necessity of addressing this vulnerability from software. WolfSSL has already provided a draft version of software fixes.

7.1 Background

7.1.1 Secure Encrypted Virtualization

AMD Secure Encrypted Virtualization (SEV) is a trusted execution environment (TEE) supported by AMD server-level EPYC processors with “Zen" Architecture. SEV aims at providing confidential virtual machines for cloud customers. In SEV’s threat model, other virtual machines, as well as the cloud host itself, are considered untrusted. The attacker may execute arbitrary code at the privileged hypervisor level and may also have physical access to the machine (*e.g.*, DRAM chips) [51]. To achieve this ambitious goal, a dedicated security subsystem consisting of the AMD Secure Processor (AMD-SP) and an AES memory encryption engine is introduced by SEV to protect data in use.

Hardware Memory Encryption. When SEV is enabled, the cryptographic isolation provided by Hardware Memory Encryption protects the confidentiality of the VM. Specifically, the VM’s memory pages are always stored in encrypted form, and the VM encryption keys are guarded by the AMD-SP. SEV adopts a 128-bit AES encryption with the XOR-Encrypt-XOR (XEX) encryption mode, which incorporates a physical address-specific *tweak* such

that the same plaintext yields different ciphertexts for each memory location. However, for a fixed address, an identical plaintext always yields the same ciphertext.

Nested Page Tables (NPT) and the page fault controlled channel. When SEV is enabled, the address translation between the VM's guest physical addresses and the host physical addresses is managed by the hypervisor with the help of a NPT, which is a two-layer page table consisting of a Guest Page Table (GPT) and a Nested Page Table (NPT). The GPT is managed inside the guest VM and thus protected by the VM encryption key. The NPT is solely managed by the hypervisor.

As shown in prior work [60, 69, 93], the hypervisor can leverage the control over the NPT to intercept the execution of the guest with page granularity. To achieve this, the hypervisor can unset the Present bit (P bit) in the NPT. The next time the VM tries to access the corresponding guest physical page, a nested page fault (NPF) will be generated, revealing the addresses of the access and the causes.

SEV extensions. Two extensions of SEV have been introduced by AMD to add additional security protections since SEV's first release in 2016.

The second generation of SEV is called SEV-ES (Encrypted State) [49], which was first introduced in 2017. SEV-ES adds additional protection for CPU registers. Prior to SEV-ES, CPU registers were stored unencrypted in the Virtual Machine Control Block (VMCB) during world switches from the VM to the hypervisor (VMEXIT). In SEV-ES, the hardware automatically encrypts the registers in a designated Virtual Machine Save Area (VMSA) along with additional integrity protection. In addition, a guest-host communication protocol was introduced for instructions that need to expose registers to the hypervisor (*e.g.*, `CPUID`, `RDMSR`, *etc.*). A VMM Communication handler (#VC handler) inside the guest VM assists the instruction emulation. Specifically, the #VC handler intercepts those instructions with

the help of hardware, passes necessary register values to a shared area called Guest-Host Communication Block (GHCB), triggers a special VMEXIT by the `VMGEXIT` instruction, and reads the resulting register values from the GHCB afterwards.

The third generation of SEV is called SEV-SNP (Secure Nested Paging) [8], which was released in 2020. As a response to attacks which used remapping or modification of guest memory in order to inject code into the VM [93], a structure called Reverse Map Table (RMP) was introduced. It maintains a second translation of host physical addresses to guest physical addresses as well as keeps track of the ownership of memory pages, and thus, prevents the hypervisor from modifying or remapping the guest VM's private memory. Most of the existing attacks against SEV and SEV-ES can be mitigated by SEV-SNP (Section **??**).

7.1.2 Ciphertext Attacks against SEV-SNP

Ciphertext attacks against SEV-SNP were first introduced by Li *et al.* in CIPHERLEAKS [62]. The work exploited leakage caused by the ciphertext of the registers inside the VMSA. Specifically, by inspecting the ciphertext stored in the VMSA during VMEXITs, an attacker could (1) infer the execution state of a known binary inside the guest VM, and (2) build a ciphertext-plaintext mapping for certain registers. For example, the ciphertext of the `RAX` register could reveal the return value of function calls. Since the ciphertext was deterministic, functions that returned the same value produced an identical ciphertext for the `RAX` register inside the VMSA, which is sufficient for the attacker to distinguish secret-related data content and steal secrets from an application using the OpenSSL library.

In response to that attack, AMD added additional randomization when encrypting and saving register values into the VMSA during VMEXITs [11]. Thus, the ciphertext of the

register state is now completely different even if the register values inside CPU did not change between two VMEXITs, which fully mitigates the CIPHERLEAKS attacks.

7.1.3 Off-chip Attacks

Off-chip attacks are usually classified into *stolen DIMM attacks* and *bus snooping attacks*. Stolen DIMM attacks directly grab data from the Non-Volatile Memory (NVM) or perform cold boot attacks on volatile memory [82]. Bus snooping attacks target the data transmission between two components of the computer (*e.g.*, CPU and DRAM). These attacks involve both data eavesdropping and even data altering [24].

Off-chip attacks are also considered as one of the potential attacks in a TEE's threat model [8]. While the plaintext is protected inside the chip and can hardly be inspected, all data outside the CPU might be inspected, either on the external memory buses or on the NVM. TEEs like Intel SGX and AMD SEV protect data outside the CPU by an in-chip memory encryption engine. While it is widely accepted that attacks by monitoring the data bus flow can be thwarted by memory encryption [86], researchers move their attention to the unencrypted address bus [24]. Recent results [57, 72] showed that an attacker could recover some data by monitoring memory address patterns. For those attacks, an interposer is needed to be installed on the DIMM socket. The interposer can duplicate signals on the memory bus and pass the data to a signal analyzer on the fly with CPU cycle granularity.

7.1.4 Operating System Context Switch

Under x86_64, there are four different privilege levels that can be used to implement a hierarchy in the software [6, Sec. 4.9.1]. Under Linux, ring 0 is used to run the kernel, while ring 3 is used to run user space applications. When a privilege level change occurs, *e.g.* due to an interrupt or exception, the CPU automatically switches to a separate stack

and fills it with some information about the previous software. The stacks are configured in the Task State Segment (TSS). The register values, however, remain unchanged and are not stored by a hardware mechanism [6, Sec. 12.2.5]. Under Linux, one TSS per CPU is used, meaning that each CPU has its own set of stacks. Most Interruptd/Exception handlers use TSS managed as an entry point to intialy store the register values, before eventually copying them to the so-called thread stack. The thread stack is part of the Process Control Block (PCB, also called `task_struct` in Linux), a data structure that bundles all information related to a process/thread. The saved registers are referred to as the `pt_regs` structure, which simply consists of the register values stored next to each other.

Note that in other scenarios a context switch is also used to describe a switch between different processes and threads. In this work, we always refer to the aforementioned privilege level change if not stated otherwise.

7.2 A generic ciphertext side channel

In this section, we are going to show that the ciphertext-based attack demonstrated in the CIPHERLEAKS paper is not limited to the VMSA register storage mechanism of SEV-SNP, but applies to any deterministically encrypted memory. We define a generic attacker model and show two primitives that allow the attacker to infer memory contents and runtime behavior of any application which relies on deterministically encrypted memory for protecting the confidentiality.

7.2.1 Attacker Model

We consider the standard threat model of confidential VM: The attacker has both software and physical access to the system, *i.e.*, they have unrestricted administrator capabilities and can physically access the machine. The confidential VM shields the VM's secrets from the

attacker by encrypting the memory consumed by the user's application, using a deterministic memory encryption scheme with an address-based tweak, such that the ciphertext depends on the encryption key, the plaintext and the current physical address. Specifically, we target SEV-SNP, which also prevents the attacker from remapping memory containing ciphertext to other physical addresses, denies them write access to any encrypted memory, but leaves the attacker the ability to read ciphertext by software.

7.2.2 Attack Primitives

We suggest two general methods for exploiting deterministic memory encryption: A *dictionary attack* and a *collision attack*.

Dictionary attack. A dictionary attack is applicable when a secret-dependent variable features a small, predictable value range with a fixed memory address. In this case, the attacker can build a dictionary of ciphertext-plaintext mappings for this variable and selectively recover the plaintext. This is a generalization of the approach taken in the CIPHERLEAKS attack, where the authors learned ciphertext mappings for the registers stored in the VMSA.

Contrary to CIPHERLEAKS, the dictionary attack targets arbitrary memory locations and variable types. Two examples about recovering ECDSA key using stack variables (Section 7.4.1), or registers stored during a context switch (Section 7.3) are presented. While this attack is quite powerful, it is restricted by the number of possible plaintexts for a given encryption block, since the attacker cannot tell which part of the plaintext has changed when observing a new ciphertext. If the targeted variable shares an encryption block with other variables which get new values frequently (*e.g.*, a loop counter), the number of possible plaintexts becomes too large to efficiently build a mapping, as is illustrated in

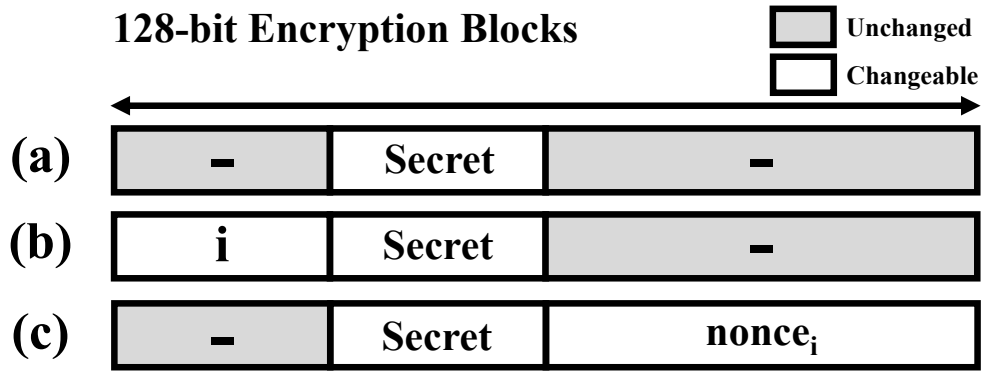


Figure 7.1: Encryption block configurations with different exploitability by the dictionary attack. In the first scenario (a), most of the block's plaintext is constant, with the secret being the only variable. Thus, the attacker can build a one-to-one mapping of ciphertexts to secrets. In (b), the block also contains a loop counter *i*, so there are many different ciphertexts mapping to the same secret. If the attacker can always observe the secret for a specific fixed value of *i*, they may still be able to build a dictionary, as this is equivalent to scenario (a). In the last scenario (c), the secret is followed by a random nonce which is regenerated before spilling secret to the memory. This prevents the attacker from creating a dictionary, as he never observes the same ciphertext twice.

Figure 7.1. We use this fact in Section 7.5.2 to propose a countermeasure which appends random nonces to small variables.

Collision attack. A collision attack transfers the concept of secret dependent code execution to memory writes. In secret-dependent branching, the attacker exploits that the targeted algorithm executes a certain code region depending on specific values of a secret value (*e.g.*, an *if* statement checking key bits). By observing the access pattern to the respective code chunks, the attacker can learn the secret. A common countermeasure is so-called *constant-time* code, *i.e.* code that always exhibits the same control flow and memory accesses, independent of the secret. This is usually achieved by converting secret-dependent branch decisions into fixed expressions, which compute all possible results of a given operation and then use a mask to pick the desired one. One such primitive is the constant time swap

Algorithm 3 Constant time swap

Require: Byte arrays a, b of same length and decision bit c

```
1: procedure CSWAP(a, b, c)
2:     mask ← 0 − c                ▷ 0 − 1 underflows to 0xff for i = 0 to i = length(a) do
3:
       end
       x ← a[i] ⊕ b[i]
4:     x ← x & mask
5:     a[i] ← a[i] ⊕ x
6:     b[i] ← b[i] ⊕ x
7:
8: end procedure
```

CSWAP (Algorithm 3), which is used for example by the Montgomery ladder: CSWAP takes two variables a and b and a (secret) decision bit c. If the bit is set, the values of a and b are swapped; if the bit is cleared, a and b remain unchanged. The depicted code gadget always executes the same amount of instructions in the same order, and always accesses the same memory addresses, making it resistant against microarchitectural side-channel attacks.

But, if the attacker is able to observe whether the values of a or b *change*, they can immediately learn the decision bit c_i. The collision attack again exploits the fact that ciphertext blocks are deterministic. However, contrary to the dictionary attack, the attacker does not aim to learn the direct mapping of ciphertexts to actual plaintext values, but they only check whether certain ciphertexts *repeat* or *change*. Going even further, if the attacker knows that a memory write was executed (*e.g.*, through a control flow side-channel), but they do not see any ciphertext change, they learn that the instruction wrote the same value as was present in memory before. Given knowledge of the executed program, they may use this to infer more information other than the traditional control flow.

7.3 Leakage due to context switch

We now take the dictionary attack primitive from Section 7.2 and show how it can be used for extracting register values from a VM running with SEV-SNP. After CIPHERLEAKS, AMD published a firmware patch which added protection to the VMSA area [11]. However, the VM-hypervisor world switch is not the only occasion where the entire register state is written to memory. When moving from user space to kernel space (*e.g.*, after an interrupt or an exception), the Linux kernel pushes all register values of the user program onto the stack, and then copies those into the PCB of the current thread, such that the exception handler can access the register values through the `pt_regs` structure. The PCB address is fixed per-thread, allowing an attacker to build a dictionary of register values by causing repeated interrupts within the VM and observing the resulting ciphertexts. We show how an attacker can use nested page faults to indirectly trigger internal user-kernel context switches and use the learned register values to attack the constant-time ECDSA implementation of OpenSSL. Given their source code, similar attacks should also be applicable in WolfSSL, GnuTLS, OpenSSH, and libgcrypt.

7.3.1 Leaking Register Values via Context Switches

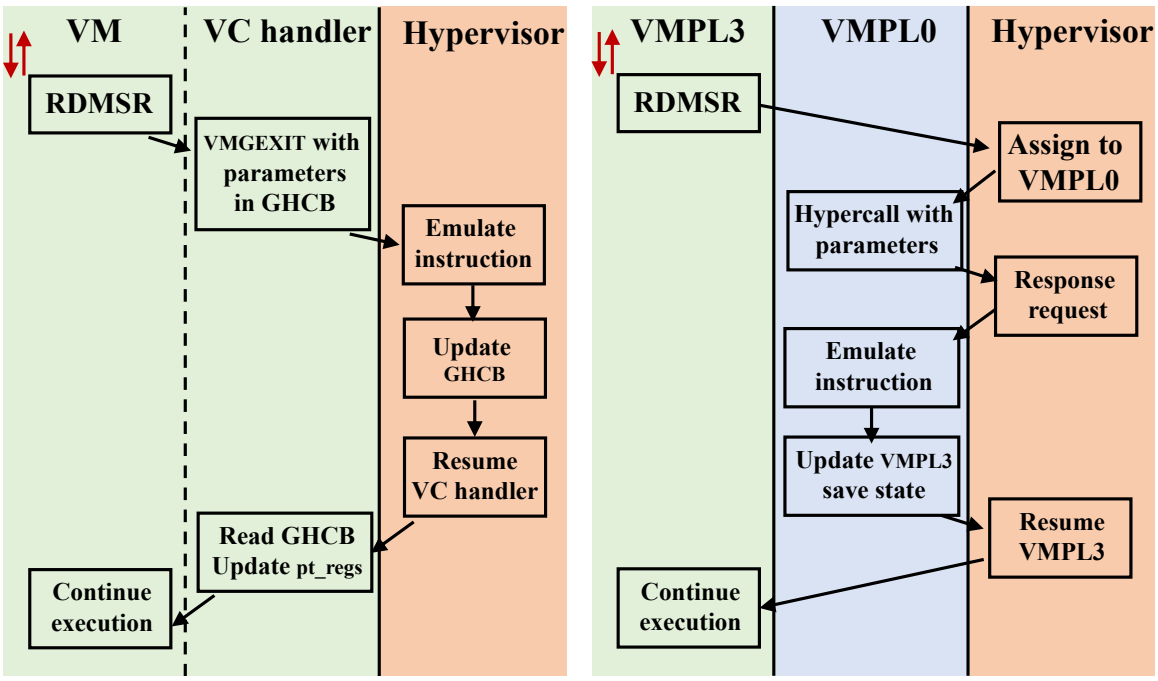
Forcing context switches in the VM. SEV-SNP restricts the hypervisor's ability to inject interrupts and exceptions into the VM, so we will show how a malicious hypervisor can work around this limitation by forcing the VM to pause at a certain execution point until a "natural" internal context switch is triggered, which should also be detectable by the hypervisor.

First, the hypervisor interrupts the targeted application at certain execution points by using the well-known page fault controlled channel, that allows the attacker to force a NPF

when the VM tries to access or execute a given page. However, the NPF itself does not lead to a context switch inside the VM, as it is immediately intercepted by the hypervisor. To do so, the hypervisor now simply waits for a short amount of time and then resumes the VM without handling the NPF. As a result, the attacker can trap the execution of the targeted program and the victim application cannot resume its execution. After a short amount of waiting time, a time-driven internal context switch will be performed by the guest OS, which updates the victim application's register values in main memory (`pt_regs`).

Even though the internal context switch is out of the hypervisor's control, we show that the VM-host interaction mechanism adopted by SEV can work as an indicator of a finished context switch. Specifically, we observed that the guest VM has frequent interaction with the hypervisor through reading and writing hypervisor-managed registers of the Advanced Programmable Interrupt Controller (APIC), like `IA32_X2APIC_TMR1`, which are used for scheduling and timekeeping. These `RDMSR` and `WRMSR` accesses result in a special exception called #VC exception inside the VM, as they require the VM to share registers with the hypervisor. The #VC exception handler inside the VM then calls `VMGEXIT` after putting the necessary register values into the GHCB (shown in Figure 7.2a). As the #VC exception is handled in VM's kernel space, a `VMGEXIT` also indicates a user-kernel context switch. Thus, the hypervisor simply waits for a `VMGEXIT` with the appropriate exit code, as an indicator of updated registers' ciphertext in `pt_regs`. We analyze the necessary pause time for triggering a `VMGEXIT` in Section 7.3.4.

Other than the traditional #VC handler mechanism, SEV-SNP has another option to adopt a more secure VM-host communication mechanism that moves the APIC emulation into the trust domain of the guest VM. As shown in Figure 7.2b, the VM is divided into multiple Virtual Machine Privilege Levels (VMPLs) that provide additional hardware isolated



(a) #VC handler (b) VMPL0 emulation

Figure 7.2: Workflow of how #VC exceptions are handled. Red arrows represent a context switch between processes.

abstraction layers. However, the hypervisor can still sense a finished context switch due to the interaction triggered by the hypercall from VMPL0.

Locating `pt_regs` after VMEXIT. Besides using the `VMGEXIT` to detect a context switch, the attacker can also use it to locate the `pt_regs` struct. For that, after reaching a `VMGEXIT`, the attacker clears the P bit for all guest pages and resumes the VM. This will hand back control to the #VC handler in the VM, which will subsequently try to copy the results of the emulated instruction from the GHCB to `pt_regs`. Since all guest pages were marked as not present, this causes a nested page fault. In our experiments, the second NPF caused by data page read access after resuming the VM is the memory page containing `pt_regs`. We did not encounter any false positives during our experiments.

7.3.2 Attacking Constant-time ECDSA

In this section, we demonstrate how to use the context switch primitive from the previous section to attack the constant-time ECDSA implementation in OpenSSL. More precisely, we show that the adversary can infer the nonce k in the constant-time ECDSA algorithm by inspecting the ciphertext changes in the `pt_regs` structure of the targeted process. This can then be used to recover the secret key.

The Elliptic Curve Digital Signature Algorithm (ECDSA) is a widely used signature algorithm that works as follows:

1. Prepare the curve parameters (CURVE, G, n), where G is the elliptic curve base point of prime order n.
2. Prepare a key pair by choosing uniform $d_A \in \mathbb{Z}_n^*$. d_A is the private key. The public key is $Q_A = d_A G$.
3. Generate a cryptographically secure random integer $k \in \mathbb{Z}_n^*$ (also known as the nonce k).
4. Calculate a non-zero r by $r = (kG)_x \mod n$ (only the x-coordinate of the resulting point is used).
5. Calculate $s = k^{-1}(h(m) + rd_A) \mod n$, where m is the message and $h(m)$ is a hash of m. (r,s) then forms the ECDSA signature pair.

A predictable or leaked nonce k allows to immediately recover the private key d_A by:

$$d_A = r^{-1}((ks) - h(m)) \mod n.$$

```
int i, cardinality_bits, group_top, kbit, pbit, Z_is_one;
...
for (i = cardinality_bits - 1; i >= 0; i--) {
    kbit = BN_is_bit_set(k, i) ^ pbit;
// kbit is used to determine the conditional swap
    EC_POINT_CSWAP(kbit,r,s,group_top,Z_is_one);
// single step of the Montgomery ladder
    if (!ec_point_ladder_step(group, r, s, p, ctx)){
        ERR_raise(ERR_LIB_EC,
        EC_R_LADDER_STEP_FAILURE);
        goto err;
    }
// pbit helps to merge CSWAP with that of the next iteration
    pbit ^= kbit;
}
```

Listing 7.1: Part of the elliptic curve scalar multiplication `ec_scalar_mul_ladder()` from OpenSSL. The function uses the Montgomery ladder algorithm and constant-time primitives to protect the secret scalar k against side channels.

Targeted ECDSA implementation. Our attack targets the ECDSA implementation of the OpenSSL library[11] for the curve `secp384r1` that is commonly used for TLS/SSL connections. The goal of our attack is to steal the nonce k and thus infer the private key d_A. In OpenSSL, ECDSA signing is handled by the `ECDSA_do_sign` function, which in turn calls `ec_scalar_mul_ladder` to calculate `r`. Note that the implementation of the function is specifically designed to protect k against side channel attacks (Listing 7.1).

Identify instruction pages. Besides monitoring context switches and locating `pt_regs` via the methods shown in the previous part, we also need to identify the appropriate code locations in order to intercept the guest VM at proper execution points, which gives the attacker the opportunity to extract valuable ciphertext. In our work, we combine the widely-used page fault controlled side channel [61, 69, 91, 93] with performance counters to build a fine-grained tool to identify instruction pages' physical addresses. Specifically, we make use of the *Retired Instructions* counter [4, Event `PMCx0C0`], which can be configured to

[11] Commit: `c4b2c53fadb158bee34aef90d5a7d500aead1f70`.

only count the amount of retired instructions inside the VM and thus reveal the number of instructions executed between two pages faults. The attacker can simply build a template of the retired instruction counts for code paths in a known binary. In our experiments, we were able to locate the target pages on the fly, without relying on repeated access patterns.

7.3.3 End-to-end attack against Nginx

We now show the steps needed to steal the nonce k generated by an Nginx webserver. The nonce, together with the corresponding signature, allows the attacker to recover the secret key of the server.

① **Send HTTPS request.** The attacker sends a HTTPS request to the Nginx server in order to trigger the targeted code paths.

② **Locate target function in physical memory.** Right after sending the HTTPS request, the attacker clears the P bit of all VM pages. The attacker then locates the guest physical addresses of the functions `ec_scalar_mul_ladder()` (gPA_0) and `BN_is_bit_set` (gPA_1) using the page fault channel combined with the retired instruction counter.

③ **Locate `pt_regs`.** The attacker pauses the VM for a while (*e.g.*, by trapping the VM in the NPF handler for a few milliseconds) when they intercept a NPF of gPA_0. They then use the method from Section 7.3.1 to find the physical address gPA_3 of the current thread's `pt_regs` structure.

④ **Single-step loop iterations.** The attacker iteratively clears the P bit of gPA_1 to pause the VM when it enters `BN_is_bit_set`. After intercepting the corresponding NPF for gPA_1, the attacker clears the P bit for gPA_0, causing an NPF when the `ret` instruction inside `BN_is_bit_set` is executed, *i.e.* the function tries to return to `ec_scalar_mul_ladder()`. The attacker then pauses the VM in the gPA_0 NPF for a while (several milliseconds) and

resumes the VM without handling the NPF. The attacker might observe several consecutive NPFs for gPA_0, but keeps the P bit cleared until a `VMGEXIT` is encountered.

⑤ **Record the ciphertext and recover the nonce k.** The attacker records the ciphertext of the RAX field in `pt_regs` after the `VMGEXIT`, which contains the return value of `BN_is_bit_set` at this execution point. The conjunct register stored near RAX in `pt_regs` is R8, which remains unchanged during the `for` loop. The attacker then sets the P bit of gPA_0, clear the P bit of gPA_1 in order to intercept `BN_is_bit_set` for the next iteration and repeat step ④. After 384 iterations, the attacker has collected a sequence of ciphertexts. Since RAX can only take two distinct values, they can recover the nonce `k` with only 1 bit of entropy.

7.3.4 Evaluation

All experiments throughout this paper were conducted on an AMD EPYC 7763 64-Core Processor. The host kernel (branch `sev-snp-part2-rfc4`), QEMU (branch `sev-snp-devel`), and OVMF (branch `sev-snp-rfc-5`) were directly forked from AMD SEV's GitHub repository [9]. The victim VMs were protected by SEV-SNP and used the unmodified guest kernel provided by AMD (branch `sev-snp-part2-rfc4`). The victim VMs were configured with 2GB DRAM, 30GB disk, and one virtual CPU (vCPU). However, the capacity of the victim VMs (including vCPU, DRAM, and disk) is not relevant for the attack procedure.

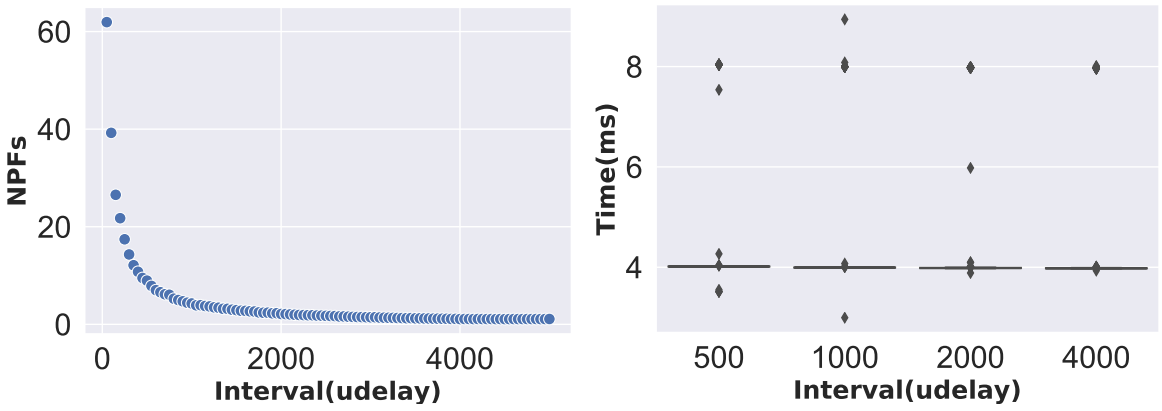
For the attack on Nginx, an unmodified Nginx server and an OpenSSL library were installed inside the victim VMs. The Nginx version is 1.21.3, which was released on 07 Sep. 2021. The Nginx server supports HTTPS requests with a self-signed ECC certificate with 384-bit key. The curve used is `secp384r1`. The OpenSSL was forked from OpenSSL's Github repository (Commit: `c4b2c53fadb158bee34aef90d5a7d500aead1f70`) and was

modified to log the ground truth after the signing procedure, so we could verify the extracted secret.

Identifying target functions. To estimate the attacker's ability to locate target functions on the fly, we sent 500 consecutive HTTPS requests. For each request, we monitored the page access pattern along with the number of retired instructions and tried to locate the target functions in real-time. The reference page access pattern and the corresponding performance counter values were collected in a different VM with the same Nginx and OpenSSL version, but without SEV-SNP's protection and with a different kernel version, to show the pattern's independence of the exact kernel version.

In 496 out of those 500 requests, the target function's physical addresses were successfully located, while a miss was reported for the remaining four requests. The average time needed to locate the target functions was 59.28 milliseconds with a standard deviation of 2.12 milliseconds. No false positive was reported.

Context-switch latency. To collect the ciphertext of the updated `pt_regs`, the attacker needs to wait until an internal context switch, which is the most time-consuming part of the end-to-end attack. In our implementation, the attacker pauses the VM by calling `udelay(<interval>)`, which takes a delay in microseconds. We evaluated both the proper interval for a direct context switch and the average waiting time. Since the attacker doesn't set the P bit at the execution point unless observing the `VMGEXIT`, the attacker might get several repeated NPFs in a row. Figure 7.3a shows the number of NPFs we observed under different intervals. We usually directly detected a context switch when `interval` was larger than 2000 (two milliseconds). Figure 7.3b shows the average waiting time. It usually took four milliseconds until an internal context switch occurred, thus we paused the victim VM by using `udelay(4000)` in our attack.



(a) #NPFs for `udelay` intervals.

(b) Avg. time for context switch.

Figure 7.3: Relationship between `udelay` interval and internal context switch.

Performance. We repeated the attack 50 times and measured the overall time for an end-to-end attack. The average time was 8.53 seconds with a standard deviation of 0.33 seconds. The main latency is caused by waiting for an interval context switch. For a 384-bit nonce k, the attacker can intercept 384 * 5 = 1920 NPFs for gPA_0 in total. In our setting, we chose to wait for a context switch every time when intercepting an NPF of gPA_0. However, for each iteration, only one out of five NPFs is caused by the `ret` instruction inside `BN_is_bit_set`. Thus, the attacker could also choose to only wait and grab ciphertext at that NPF. By doing that, approximately 6 seconds (384 * 4 * 4ms) waiting time can be avoided. However, one side effect is that some internal events (*e.g.*, an unexpected context switch) might cause a repeated NPF of gPA_0, which will confuse the attacker and reduce the accuracy. In our implementation, the average accuracy for the recovered nonce `k` is 89.1%.

7.4 Exploiting memory accesses in user space

In the previous section, we have seen how an attacker can exploit the context switch mechanism of the Linux OS inside the VM to leak register values of running processes.

We now turn our attention to leakages directly caused by the victim application's memory access behavior. We demonstrate that the OpenSSL ECDSA code from the previous section is also vulnerable to the dictionary attack targeting stack variables, and show an example of the collision attack against the EdDSA implementation in OpenSSH.

7.4.1 Breaking Constant-time ECDSA via Dictionary Attack

As shown in Listing 7.1, `ec_scalar_mul_ladder` uses several local integer variables: `kbit` controls the conditional swaps by `EC_POINT_CSWAP` in the `for` loop. Assuming that k_i refers to the i-th bit of k, at the beginning of a loop iteration, `pbit` stores k_{i-1}. After calling `BN_is_bit_set(k, i)` to retrieve k_i, `kbit` stores $k_{i-1} \oplus k_{i-2}$ (XOR). `pbit` is later updated to k_i at the end of the iteration.

Stack layout. We target the 16-byte memory block where `pbit` is stored. By our observation, the memory block containing `pbit` also contains additional variables, which is not surprising given the small size of `pbit`. In our case, `pbit`, `kbit` and `cardinality_bits` all share the same 16-byte memory block. The `cardinality_bits` variable does not change during the runtime of the `for` loop from Listing 7.1. Thus, the value range of the ciphertext is only dependent on the secret, *i.e.* `pbit` and `kbit`.

Recovering `k` from ciphertext pairs. Recall that, at the end of each loop iteration, `pbit` stores the i-th bit of the nonce k. The attacker thus can recover k if they can infer the value of `pbit` in each iteration. We use gPA_0 to denote the guest physical address of the stack page where `pbit` is stored, and gPA_1 for the address of `BN_is_bit_set()`. Similar to the attack in Section 7.3.1, the attacker uses the page fault controlled channel in combination with the retired instructions performance counter for locating the pages.

The attacker records the ciphertext of gPA_0 when he intercepts the NPF of `BN_is_bit_set()` (gPA_1), which corresponds to the state after the previous loop iteration (*i.e.*, `pbit` still has its old value). As shown in Table 7.1, in the i^{th} iteration, the attacker can observe one of four possible `pbit` and `kbit` pairs. We use the letters `A` to `D` to denote the four possible ciphertexts. At the end of the i-th iteration, `pbit` and `kbit` are updated according to k_i (0 or 1). Thus, when the attacker intercepts the NPF of gPA_1 in the $i+1$-th iteration, there are 8 possible observation cases.

They then analyze the ciphertext of gPA_0 to (1) locate the offset of the 16-byte block where `pbit` is in and to (2) infer the value of `pbit` for this iteration. For (1), the attacker can easily identify the offset because they should observe the four different ciphertext randomly but repeatedly at a certain offset, which reveals the ciphertext changes of the pair (`pbit`, `kbit`). For (2), the attacker can infer the value of `pbit` by analyzing two subsequent ciphertext of (`pbit`, `kbit`) as shown in Table 7.1. The attacker applies the following algorithm to recover the `pbit` sequence: In the first iteration, both `kbit` and `pbit` are initialized to 1, thus producing ciphertext `D`. The attacker then finds an n-th iteration that has the same ciphertext as the following $n+1$-th iteration. Then (`pbit`, `kbit`) for the n-th and $n+1$-th iterations must either be `A` or `C`. If the next $n+x$-th iteration with a different ciphertext produces a ciphertext other than `D`, then the ciphertext for n^{th} and $n+1^{th}$ iterations must be `C`. Otherwise, the ciphertext represents `A`. After identifying `A`, `C`, and `D`, the remaining ciphertext represents `B`.

7.4.1.1 Attack Steps

Table 7.1: Possible `pbit` and `kbit` pairs when intercepting `BN_is_bit_set()`. The letters `A` to `D` represent the 16-byte ciphertexts the attacker may observe, which depend on the values of `kbit` and `pbit`. The value of `kbit` and `pbit` in the $i+1$-th iteration is updated depending on k_i.

***i*-th iteration**				***i*+1-th iteration**		
`pbit`	`kbit`	**Pair**	k_i	`pbit`	`kbit`	**Pair**
0	0	A	0	0	0	A
0	0	A	1	1	1	D
0	1	B	0	0	0	A
0	1	B	1	1	1	D
1	0	C	0	0	1	B
1	0	C	1	1	0	C
1	1	D	0	0	1	B
1	1	D	1	1	0	C

① **Locate the two target physical addresses.** The attacker first needs to locate the guest physical addresses of the target stack page gPA_0 and the target function page gPA_1 . We use the same methods as in Section 7.3.1 to locate the pages.

② **Intercept the `for` loop.** The attacker iteratively clears the P bit in the NPT to interrupt the execution of the `for` loop. Specifically, the attacker clears the P bit of gPA_0 when a NPF of gPA_1 is intercepted and clears the P bit of gPA_1 when a NPF of gPA_0 is intercepted later. The attacker thus tracks the internal execution states of the `for` loop.

③ **Record the ciphertext of gPA_0.** Given the structure of the loop, there are 5 NPFs for both gPA_0 and gPA_1 for one iteration. Thus, for a 256-bit nonce k, the attacker needs to intercept 256 * 5 = 1280 NPFs for both gPA_0 and gPA_1. In each iteration, the first NPF for gPA_0 is triggered when `BN_is_bit_set` finishes execution and the program tries to touch the stack page where (`pbit` and `kbit`) is in. At this execution point, both `kbit` and the

`pbit` are not yet updated. The attacker records the ciphertext of the whole stack page since the offset of `pbit` and `kbit` change slightly between different runs of the algorithms.

④ **Infer the value of *k*.** After all 256 iterations of the `for` loop, the attacker determines the offset and recovers the nonce *k* using the strategy we introduced in Section 7.4.1.

7.4.1.2 Evaluation

The test platform was the same as described in Section 7.3.4. Instead of targeting the `secp384r1` curve, we picked a different curve `secp256k1`, which is widely used in Bitcoin, to show that the attack works for different curves. The victim VM computes an ECDSA signature by calling `ECDSA_do_sign` in the OpenSSL library. We repeated the attack 50 times. In 92% of the attempts, we could recover the nonce `k` with 100% accuracy. After identifying the target functions, which we only needed to do once, the average time used to conduct the attack is 1.23 seconds with a standard deviation of 1.01 seconds.

7.4.2 Breaking Constant-time EdDSA via collision attack

In the previous attack case studies we have used the dictionary attack primitive by guessing and recording plaintext-ciphertext mappings. We now show how the attacker can break constant-time EdDSA by monitoring the collision of the secret dependent value's ciphertext. While the attack would also be applicable to the constant time swaps used by the ECDSA variant described above, we show how the collision attack can work on the constant time EdDSA implementation of OpenSSH with the `ed25519` curve. As this implementation processes the secret in a batched manner, it is less susceptible to the dictionary attack previously applied to the ECDSA implementations.

The EdDSA signature algorithm [19] works similar to ECDSA, with the most noticeable difference being the deterministic nonce generation to prevent attacks based on flawed random number generators. The algorithm works as follows:

1. Provide a valid EdDSA parameter set (CURVE, G, n, c, l, H) with $2^c \cdot l = |\text{CURVE}|$, where G is the elliptic curve base point of prime order l and thus $l \cdot G = 0$. H is a cryptographic hash function with $2b$ output bits.
2. Prepare a key pair. Choose a secure random b-bit string d_A as the secret key. Calculate the public key $Q_A = d_s G$, where d_s is derived from the hash of d_A.
3. Deterministically compute a nonce for the signature as $k = H(H_{b,\ldots,2b-1}(d_A) \,\|\, m)$, where m is the message.
4. Calculate $R = kG$.
5. Calculate $s = k + H(R \,\|\, Q_A \,\|\, m) \cdot d_s \mod l$. The final EdDSA signature is defined as the tuple (R, s).

Targeted EdDSA implementation. We target the EdDSA implementation of OpenSSH 8.2p1, which is the version shipped with the latest Ubuntu LTS 20.04. The targeted implementation uses the `ed25519` curve. More precisely, we attack the multiplication $R = kG$ to learn k which then allows us to recover d_s from s, by computing

$$d_s = (s - k) \cdot H(R \,\|\, Q_A \,\|\, m)^{-1} \mod l.$$

While d_s is not the actual private key d_A, it is sufficient to create valid signatures.

Listing 7.2 shows the function performing the calculation $k \cdot G$. The arithmetic is implemented using a windowing technique with pre-computed partial sums in a lookup table. First, in line 6, the secret scalar is broken down into 3-bit chunks. In addition, a transformation is applied converting the chunks to signed values. However, this is reversible.

```
1  void ge25519_scalarmult_base(ge25519_p3 *r, const sc25519 *k) {
2    signed char b[85];
3    int i;
4    ge25519_aff t;
5    sc25519_window3(b,k);
6    choose_t((ge25519_aff *)r, 0, b[0]);
7    fe25519_setone(&r->z);
8    fe25519_mul(&r->t, &r->x, &r->y);
9    for(i=1;i<85;i++) {
10     choose_t(&t, (unsigned long long) i, b[i]);
11     ge25519_mixadd2(r, &t);
12   }
```

Listing 7.2: Function performing the multiplication of the secret scalar with the curve base point. In the original code, the variable k is named s.

Lines 12 and 13 in the for loop contain the main multiplication work. In `choose_t` the partial sum is loaded from the precomputation table in a cache attack resistant manner by accessing multiple values and choosing the correct one using a constant time swap operation. Line 13 performs the actual multiplication.

For our attack, we focus on the constant time swap operation `cmov_aff` that is used in `choose_t`. Both functions are shown in Listing 7.3. The idea of the attack is to use the collision attack to leak the value of `b`, which corresponds to d_s in our EdDSA description, in the calls to `cmov_aff`. We compare the values of `t` before and after the function call. While the constant-time swap will write to the memory locations regardless of the value of `b`, to be secure against cache and timing side channels, the actual value that is written still depends on `b`. Although the written data has a large value range, making a dictionary attack infeasible, it suffices to compare the ciphertext of `t` before and after the call to `cmov_aff` without knowing the plaintext for the ciphertext. The information whether the ciphertext value has changed or not allows us to directly infer `b`.

After leaking the value of `b`, the attacker can invert the operations applied in `sc25519_window3` (Listing 7.2) to recover the secret scalar k. Knowing k and the corresponding signature (R,s)

```
static void cmov_aff(ge25519_aff *r, const ge25519_aff *p, unsigned
    char b) {
     fe25519_cmov(&r->x, &p->x, b);
     fe25519_cmov(&r->y, &p->y, b);
}

static void choose_t(ge25519_aff *t, unsigned long long pos, signed
    char b) {
  fe25519 v;
  int i = 0;
  *t = ge25519_base_multiples_affine[5*pos+0];
  cmov_aff(t, &ge25519_base_multiples_affine[5*pos+1],equal(b,1) |
    equal(b,-1));
  cmov_aff(t, &ge25519_base_multiples_affine[5*pos+2],equal(b,2) |
    equal(b,-2));
  cmov_aff(t, &ge25519_base_multiples_affine[5*pos+3],equal(b,3) |
    equal(b,-3));
  cmov_aff(t, &ge25519_base_multiples_affine[5*pos+4],equal(b,-4));
  fe25519_neg(&v, &t->x);
  fe25519_cmov(&t->x, &v, negative(b));
}
```

Listing 7.3: Swap and lookup table access functions.

allows to recover d_s, which is sufficient to create arbitrary valid signatures. Knowing d_s is not equal to knowing the secret key d_A, as the latter is still required to compute the nonce k according to step 3. However, only a party knowing the *private* key d_A can detect this subtle difference.

7.4.2.1 Attack Steps

① **Trigger the OpenSSH server.** The attacker opens an SSH connection with the server, and explicitly requests the usage of the EdDSA key. EdDSA is enabled in the default configuration under Ubuntu.

② **Locate the target physical addresses.** The attacker uses the page fault controlled channel and the performance counter technique from Section Section 7.3.1) to infer the physical addresses of the `choose_t` and `fe25519_cmov` functions.

③ **Intercept execution before and after the constant time swap operation.** The attacker then uses the page fault controlled channel to intercept the execution of the VM by unsetting the P bit of the targeted pages in the NPT.

④ **Take snapshots of the buffer `t`.** The attacker obtains the physical address of the buffer `t` by tracking the write access pattern during the execution of the constant time swap operation using the NPF side channel. The attacker then steps the loop using the page fault controlled channel and takes snapshots of the buffer `t` in each iteration.

⑤ **Recover the secret scalar `t`.** Using the snapshots of the buffer `t` before and after each call to `fe25519_cmov` in `choose_t` (note that `cmov_aff` wraps this function), the attacker can immediately deduce the value of `b`. After knowing the value of `b`, the attacker inverts the windowing and sign transformation operations applied in `sc25519_window3(b,s)` to obtain the secret scalar k. The attacker uses the first parameter R of the signature that the server sends in step ① to validate the value of k, and extracts the signing secret d_s from the second parameter S of the signature using k.

7.4.2.2 Evaluation

We ran the end-to-end attack 500 times. In 86% of the attacks, we could fully recover the signing secret with 100% accuracy. Of the failed attack runs, only 7 where due to errors in detecting the correct code pages. The remaining errors are most likely misdetections of the memory location of the buffer `t`. The average runtime of the attack was 7.9 seconds with 2.2 seconds standard deviation.

7.5 Countermeasures

There are two categories of countermeasures against the attacks presented in this paper: First, the underlying issue may be addressed at the architectural level, which would likely be the most reliable approach. Otherwise, the identified problems can be also tackled at the software level, with a certain performance overhead. We discuss both hardware/architecture-based and software-based countermeasures, and point out methods for hardening existing software against the attacks presented in this paper.

7.5.1 Architectural Countermeasures

There are two possible hardware approaches for closing the ciphertext side channels. However, both approaches introduce high overhead.

First, one may change the encryption mode of SEV to use *probabilistic* encryption: a random nonce or incremental counter is included in the encryption and is updated on each memory write, effectively randomizing the resulting ciphertexts on each write. However, probabilistic memory encryption requires additional memory for storing the nonces. For example, Intel SGX combines AES-based probabilistic encryption with MACs to achieve confidentiality, integrity and replay protection. In SGX, data is encrypted in a tweaked counter mode, where the nonce depends on both the physical address of the encrypted memory block and a 56 bit counter value, to ensure replay protection [37]. The counter values are kept in the integrity tree, together with the MAC tags that ensure integrity protection. Only the head nodes of the tree are stored on-chip, while the remaining integrity tree remains in memory and needs to be checked on each memory access, resulting in a significant memory and latency overhead.

A second approach is preventing the attacker from reading the VM's physical memory: On a software/firmware layer, this could be achieved by using a similar RMP mechanics as in SEV-SNP (Section 7.1.1), which already prevents write accesses through an additional RMP check. However, this would introduce a certain overhead when applied to all read operations due to the more frequent read access and the extra RMP lookup. For example, for a single read access inside the VM, a series of RMP checks are needed, including four checks for the 4-level GPT and one check for the data page. For each GPT level, four additional RMP checks are needed for the 4-level NPT. In addition, on-chip access control may still be susceptible to the off-chip attacks described in Section 7.1.3.

7.5.2 Software-based Countermeasures

While hardware-based countermeasures would be preferable due to stronger security guarantees, their feasibility and practicality demand further validation. Thus, in the following sections, we describe general methods for mitigating the vulnerabilities on a software level. There is no single software-based method that is perfectly suited for all scenarios, as kernel structures, stack, and heap are all vulnerable. Thus, we present how applications can mitigate ciphertext side channels in three different ways, building on the assumption, that register values are immune to the ciphertext side channel. However, as shown in Section 7.3, this is not the case, as the kernel stores the registers' content in memory upon context switches. Thus, we also present how the ciphertext side channel caused by register states stored inside kernel structures can be mitigated with a kernel patch, to achieve the invariant of secure registers (Section 7.5.3), and measure the kernel patch performance (Section 7.5.4).

Secret-aware register allocation. If secret-related variables would fit into a register, but are kept in memory due to register pressure, changing the register allocation strategy may be

worth pursuing. The secret-related variables can be protected by staying inside the register during their lifecycle and never being spilled to memory.

In order to do that, compiler-level modifications are needed. Even though developers can suggest the compiler to keep some variables into registers by applying a `register` hint (*e.g.*, `register int var;`), the variables are not guaranteed to be placed inside registers. Thus, a compiler can be modified to prioritize variables marked as 'secret' when allocating registers. An example of a similar scheme is GINSENG [99], which employs a custom register allocation strategy and a secure storage in a TEE to shield sensitive variables from a malicious operating system. In case a register containing a secret must be spilled to the stack anyway (*e.g.*, it is frequently used in function calls or large variables), it can be protected using a random mask as described in the later software-based probabilistic encryption part.

Limiting reuse of memory locations. Both the dictionary attack and the collision attack rely on repeated writes to a fixed physical memory address. Thus, limiting reuse of a fixed memory address leads to fresh ciphertext and can prevent the attacker from inferring secrets via the ciphertext.

To achieve this, the application developer has to identify and rewrite vulnerable code sections. For example, in our collision attack (Section 7.4.2), the conditional swap operation should not be written to be performed in-place, but should store the result in a newly allocated memory area. In this way, an attacker always observes a fresh ciphertext in a new location, independent from the value of the decision byte c_i.

Software-based probabilistic encryption. If the aforementioned methods are not applicable, one can mimic probabilistic encryption in software and add a random nonce to the secret data each time when the data is written to the memory.

This can be approached in two ways: First, one can modify the memory layout of the affected data structures to include random nonces in between, such that each memory block gets a sufficient amount of random bits. Second, the memory layout is left as-is, but a second buffer of the same size is allocated for storing masks, which are then XOR-ed onto the plaintext.

The first approach can be implemented by reserving the high 8 bytes of each 16-byte encryption block for a random nonce, while the low 8 bytes are used for payload. When storing a value in this block, the nonce is incremented to ensure that the ciphertext changes. In addition, the old plaintext must be overwritten with a random value before storing the new plaintext, to keep the attacker from detecting consecutive writes of the same value. In the second approach, the nonces and the data are stored in separate locations, and the nonces are XOR-ed onto the data as a mask. On each memory write, the corresponding location in the mask buffer is resolved, the mask value is updated and then XOR-ed to the new plaintext. Finally, the masked plaintext is written to the desired memory address. As the nonces are high entropy values and updated independently of the written data, they are not susceptible to the dictionary attack or collision attack. Due to its high locality, the first approach is better suited for small variables (*e.g.*, variables on the stack), while the second approach has better support for pointer arithmetic and should thus be used for buffers and complex data structures. Both countermeasures could be implemented as a compiler extension, that automatically applies them to variables marked as secret.

7.5.3 Software-based Countermeasures: Kernel Context Switch

While the generic software-based countermeasures are sufficient to protect applications in user mode, they make the critical assumption that registers are immune to ciphertext

side channels. However, our attack in Section 7.3 shows that the attacker can inspect the ciphertext in the kernel's `pt_regs` structure to infer register values. To mitigate the ciphertext leakage on register-level, we developed a kernel patch that protects registers during context switches. We focus on the Linux kernel, but similar methods can also be applied to other operating systems.

Specifically, the kernel patch protects the `pt_regs` structure, which stores x86-64 user space registers as described in Section 7.1.4. We present two methods for securing this structure. One is to insert a random nonce alongside each register. The other is to randomize the stack location on each context switch.

Storing a nonce alongside registers. A random 64 bits nonce can be stored next to each register (64-bit) to add enough randomization. In this way, on a context switch, the kernel doesn't simply push all registers to the stack, but interleaves them with pushes of a random value, which is incremented on every context switch. This method gives us 64 bits of security, which makes it impossible for the attacker to infer the plaintext even for long running VMs. However, this strategy comes with a major caveat: It requires significant changes to existing highly-optimized code paths, as a lot of exception/signal handling functions rely on the exact offset of the registers in `pt_regs` and would thus may not be adapted by the upstream kernel committee.

Context switch stack randomization. As an alternative strategy, we adapt the memory address randomization idea to the kernel entry point stack. Instead of inserting nonces between the saved registers, we randomize the address of the stack where the exception/interrupt handlers store the register values of the interrupted user space application.

This method is much less intrusive than the nonce approach and easy to hide behind a feature flag, as we only need to keep track of stack pages and replace the stack pointer

on each exit from kernel space to user space. However, it also comes with a high memory overhead, as we have to reserve a lot physical memory only for the kernel entry point stacks. Also, at some point we will run out of physical memory, giving us a hard limit on the reachable entropy.

For example, if we assume that we have 8 GB of physical memory which can be freely used for our stack countermeasure, with a stack size of 4 KB (one page) we get 2^{21} possible stack locations (21 bits of entropy). This is significantly less than the 64 bits obtained with the nonce approach, but still considerably reduces the attack bandwidth, as the attacker would have to wait until a stack page repeats. To assess the practicality and the resulting overhead, we implemented the stack randomization countermeasure in the Linux kernel.

7.5.4 Case Study: Randomizing pt_regs Location

For our case study, we focused on the common exception and interrupt path described by `idtentry_body` which is defined in `arch/x86/entry/entry_64.S`. The `idtentry_body` path is *e.g.* used for the high frequency page fault exception as well as for the local APIC timer interrupt. The latter is especially interesting, as it is the main driver in determining if a task has used up its time slice, leading to a reschedule to a different task. While interrupts and exceptions can also occur when the CPU is already in kernel mode, we restrict our countermeasure to events that interrupt a user space application, as they contain the register values that we want to protect.

Since the thread stack is empty upon entering the kernel from user space, we can simply replace it with a newly allocated stack. For the entry stack, randomizing the stack upon entry to the kernel is more difficult, as all general purpose registers hold user data and thus cannot

be used to perform the change. To circumvent this, we randomize the stack on the exit path before returning back to user space. Thus upon the next entry, we have a fresh entry stack.

Using the regular memory allocation mechanisms of the Linux kernel for the stack allocation proves difficult, as they were not build with guarantees regarding not returning a recently freed page upon a new allocation. In addition, they share a common memory pool with the rest of the system, which increases the collision probability under high memory load, if taking random pages from the pool. Instead we allocate a large chunk of memory at boot time and manage the stacks in a first-in-first-out queue, maximizing the time between reuses.

To evaluate the performance of our prototype implementation, we call the `cpuid` instruction 10 million times in a tight loop from a user space application. Under SEV, this is an emulated instruction that will directly trigger the modified code paths in `idtentry_body` without doing further expensive computations, allowing us to efficiently measure the performance impact of the modifications to the context switch. Using this strategy, we measured a total average overhead of 1063 nanoseconds per context switch with standard derivation 4.93. We also ran a modified benchmark, where the application also loops over a large memory buffer each iteration, to measure the additional cache pressure created by randomizing the kernel stack. We ran the experiment 1000000 times resulting in a total average overhead of 2232 nanoseconds with standard derivation 297.

7.6 Discussion

Secure encryption of large memory. Memory encryption is a basic building block used in TEEs to establish the confidentiality of data that leaves the CPU. Ideally, a probabilistic

authenticated encryption scheme needs to be used, as was implemented for the first generation of Intel SGX [37]. However, managing and updating authentication tags and counter values consumes additional storage, costs latency and decreases the memory bandwidth for payload data. Thus, we do not believe that integrity trees can scale to protect large amounts of memory, as it is required for the confidential VM usage model.

To cope with these conflicting properties, many confidential VM designs use a mixture of cryptography and additional, architectural permission checks to achieve their security guarantees. Since random memory access latency is a critical performance property for the entire system, ECB would be the best candidate from a performance point of view. However, the independent encryption of all memory blocks with the same key leaks repetition patterns, as there is only one ciphertext for each plaintext. Thus, current confidential VM designs (AMD SEV [51]), but also designs to be commercially available in the near feature (Intel TDX [44] and ARM CCA [13, 14]) all adopt a tweaked block cipher, like AES XTS/XEX. Table 7.2 shows a more comprehensive overview. These modes offer a middle ground between performance and security, as the tweak mechanism offers a cheap way to ensure that the same plaintext encrypts to different ciphertexts when stored in two different addresses. However, for a given memory block, there is still only one ciphertext for each plaintext. As we have seen throughout this paper, this is the root cause of the ciphertext side channels.

To prevent attacks on the missing integrity protection, systems like SEV-SNP or Intel TDX and Intel SGX prevent untrusted parties from writing to protected memory [8,27]. Intel TDX and SGX also prevent read accesses to the ciphertext [27,44]. However, as discussed in Section 7.1.3, these checks do not prevent physical attacks like bus snooping.

Finally, the implementation of access right checks also comes with technical hurdles. On the one hand, they need to be fast, as they influence the memory access latency. On the

other hand, static approaches that simply block access to a fixed range, like in Intel SGX, hinder efficient memory use and scaling. These hurdles remain open research questions to be answered in the future works.

Side-channel resistant cryptosystems. With decades of studies on micro-architectural side channels, including cache or TLB side channels, building side-channel resistant cryptographic implementations has become a common practice. Most practically used cryptographic libraries adopt some levels of side-channel defenses, to prevent exploitation from a remote attacker [1] or another user on shared machines [100, 101]. The known best practice for defeating side channels is data-oblivious constant-time implementation, which dictates the execution time of the cryptographic operations (or an arbitrary portion of it) is constant regardless of the secret values used in the computation and that branch decisions or memory accesses may not depend on secret values. Data oblivious Constant-time implementation has been shown to defeat all known micro-architectural side-channel attacks, except the ciphertext side-channel attacks discussed in this work.

The ciphertext side channel opens up a new way of exploiting cryptographic code, which the data oblivious constant-time implementation is no longer sufficient to guard against. Given the difficulties of securing accesses to the ciphertext through memory access or bus snooping (Section 7.1.3), we envision cryptographic code to be used in TEEs with large memory needs to adopt a new paradigm that achieves indistinguishability not only on execution time and access patterns, but on the ciphertext values. We hope our work will inspire a new research direction on secure implementation of cryptography, such as tools to automate the discovery of such vulnerabilities, compilers to transform a vulnerable code to a secure one, or formal provers to assert the absence of such vulnerabilities.

Table 7.2: Comparison of hardware memory encryption-based TEEs. Drop-In replacement means that applications do not need to be adjusted to work with the TEE. * denotes the release time of the whitepapers while the commercial machine is not available yet. † denotes the new SGX version (SGX on Ice Lake SP).

Project	Vendor	Release	TCB type	TCB size	Encryption mode	Block size
SEV [51]	AMD	2016	VM	No Limit	XE or XEX	128-bit
SEV-ES [49]	AMD	2017	VM	No Limit	XE or XEX	128-bit
SEV-SNP [8]	AMD	2020	VM	No Limit	XEX	128-bit
SGX [27]	Intel	2015	Enclave	256 MB [43]	AES-CTR	128-bit
SGX† [45, 47]	Intel	2021	Enclave	up to 1 TB	XTS	128-bit
TDX [44]	Intel	*2020	VM	No limit	XTS	128-bit
CCA [13]	ARM	*2021	VM	No limit	AES XTS	128-bit

7.7 Summary

In this chapter, we have performed a comprehensive study on the ciphertext side channels. Our work extends ciphertext side-channel attack to exploit the ciphertext leakage from *all* memory pages, including those for kernel data structures, stacks and heaps. We have also proposed a set of software countermeasures, including patches to the OS kernel and cryptographic libraries, as a workaround to the identified ciphertext leakage.

As a general design lesson, deterministic encryption modes such as XEX must be combined with both read and write protection to prevent software-based attacks. To also prevent physical memory attacks, freshness and integrity protection are required.

Chapter 8: Conclusion

This dissertation systematically goes through the design flow of AMD SEV, a VM-based hardware-protected TEE. To protect SEV-protected VMs against an untrusted cloud service provider, SEV adopts some additional designs atop traditional Virtualization. Some of those adjustments are challenged, including *AES memory encryption*, the *Nested Page Table* and *Context-switch*. Thus, some designs inherited from AMD's traditional hardware-based virtualization are shown to be insecure under the assumption of the untrusted host in this dissertation, including *I/O security*, *ASID-based key management*, *ASID-tagged TLB entries*, *state store or load during context-switch*, and *deterministic memory encryption*.

Bibliography

[1] Nadhem J Al Fardan and Kenneth G Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy*, pages 526–540. IEEE, 2013.

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *25th USENIX Security Symposium*, pages 53–70, 2016.

[3] AMD. AMD-V nested paging. `http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf`, 2008.

[4] AMD. Open-Source Register Reference For AMD Family 17h Processors Models 00h-2Fh. *Manual*, July 2018. Rev 3.03.

[5] AMD. Solving the cloud trust problem with WinMagic and AMD EPYC hardware memory encryption. *White paper*, 2018.

[6] AMD. AMD64 architecture programmer's manual volume 2: System programming. *Manual*, 2019.

[7] AMD. SEV API version 0.22, 2019.

[8] AMD. AMD SEV-SNP: Strengthening VM isolation with integrity protection and more. *White paper*, 2020.

[9] AMD. AMDSEV/SEV-ES branch. `https://github.com/AMDESE/AMDSEV/tree/sev-es`, 2020.

[10] AMD. SEV secure nested paging firmware API specification. *API Document*, 2020.

[11] AMD. AMD Secure Encryption Virtualization (SEV) Information Disclosure (Bulletin ID: AMD-SB-1013). `https://www.amd.com/en/corporate/product-security/bulletin/amd-sb-1013`, 2021.

[12] AMD. AMD Virtualization (AMD-V). `https://www.amd.com/en/technologies/virtualization-solutions`, 2021.

[13] ARM. Arm CCA Security Model, August 2021. Rev 1.0, Document Number DEN0096.

[14] ARM. Arm Confidential Compute Architecture software stack. `https://developer.arm.com/documentation/den0127/latest`, 2021.

[15] Amazon AWS. Optimizing latency and bandwidth for AWS traffic, 2016.

[16] Antonio Barresi, Kaveh Razavi, Mathias Payer, and Thomas R Gross. *cain*: Silently breaking *aslr* in the cloud. In *9th USENIX Workshop on Offensive Technologies*, 2015.

[17] BearSSL. Why constant-time crypto? `https://www.bearssl.org/constanttime.html`, 2021.

[18] Michael Becher, Maximillian Dornseif, and Christian N. Klein. FireWire: all your memory are belong to us. In *CanSecWest*, 2005.

[19] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, volume 6917 of *Lecture Notes in Computer Science*, pages 124–142. Springer, 2011.

[20] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. The guard's dilemma: Efficient code-reuse attacks against intel SGX. In *27th USENIX Security Symposium*, pages 1213–1227. USENIX Association, 2018.

[21] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies*, 2017.

[22] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[23] Robert Buhren, Shay Gueron, Jan Nordholz, Jean-Pierre Seifert, and Julian Vetter. Fault attacks on encrypted general purpose compute platforms. In Gail-Joon Ahn, Alexander Pretschner, and Gabriel Ghinita, editors, *Proceedings of the Seventh ACM Conference on Data and Application Security and Privacy, CODASPY 2017, Scottsdale, AZ, USA, March 22-24, 2017*, pages 197–204. ACM, 2017.

[24] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Sgx-step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution, SysTEX@SOSP 2017, Shanghai, China, October 28, 2017*, pages 4:1–4:6. ACM, 2017.

[25] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, et al. Fallout: Leaking data on meltdown-resistant cpus. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 769–784, 2019.

[26] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. Sgxpectre attacks: Stealing intel secrets from sgx enclaves via speculative execution. In *4th IEEE European Symposium on Security and Privacy*. IEEE, 2019.

[27] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, page 86, 2016.

[28] CTS. Severe security advisory on AMD processors. `https://safefirmware.com/amdflaws_whitepaper.pdf`, 2017.

[29] Zhao-Hui Du, Zhiwei Ying, Zhenke Ma, Yufei Mai, Phoebe Wang, Jesse Liu, and Jesse Fang. Secure encrypted virtualization is unsecure. *arXiv preprint arXiv:1712.05090*, 2017.

[30] Zhao-Hui Du, Zhiwei Ying, Zhenke Ma, Yufei Mai, Phoebe Wang, Jesse Liu, and Jesse Fang. Secure encrypted virtualization is unsecure. *CoRR*, abs/1712.05090, 2017.

[31] Fujian Chuang YI Jia He Digital Inc. Anjian v1.1.0. `www.anjian.com`, 2019.

[32] Blaise Gassend, G. Edward Suh, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Caches and hash trees for efficient memory integrity verification. In *9th International Symposium on High-Performance Computer Architecture*, 2003.

[33] Google. Introducing google cloud confidential computing with confidential VMs. `https://cloud.google.com/blog/products/identity-security/introducing-google-cloud-confidential-computing-with-confidential-vms`, 2020.

[34] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on intel sgx. In *EUROSEC*, 2017.

[35] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. Aslr on the line: Practical cache attacks on the mmu. In *NDSS*, volume 17, page 26, 2017.

[36] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ Flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[37] Shay Gueron. A memory encryption engine suitable for general purpose processors. *IACR Cryptol. ePrint Arch.*, page 204, 2016.

[38] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. In *USENIX Annual Technical Conference*, 2017.

[39] Felicitas Hetzelt and Robert Buhren. Security analysis of encrypted virtual machines. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2017, Xi'an, China, April 8-9, 2017*, pages 129–142. ACM, 2017.

[40] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *2013 IEEE Symposium on Security and Privacy*, pages 191–205. IEEE, 2013.

[41] Amr Hussam Ibrahim, Mohamed Bakr Abdelhalim, Hanadi Hussein, and Ahmed Fahmy. An analysis of x86-64 instruction set for optimization of system softwares. *Planning perspectives*, page 152, 2011.

[42] Intel. Intel architecture memory encryption technologies specification, April 2019. Ref:336907-002US, Rev: 1.2.

[43] Intel. 10th Generation Intel Core Processor Families. `https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/10th-gen-core-families-datasheet-vol-1-datasheet.pdf`, July 2020.

[44] Intel. Intel Trust Domain Extensions. *Whitepaper*, 2020.

[45] Intel. Product brief, 3rd gen intel xeon scaleable processor for iot. `https://www.intel.com/content/www/us/en/products/docs/processors/embedded/3rd-gen-xeon-scalable-iot-product-brief.html`, 2021.

[46] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 380–392, 2016.

[47] Simon Johnson, Raghunandan Makaram, Amy Santoni, and Vinnie Scarlata. Supporting intel sgx on multi-socket platforms. *Intel Corporation.[Online]. Available: https://www. intel. com/content/www/us/en/architecture-and-technology/software-guard-extensions/supporting-sgx-on-multi-socket-platforms. html*, 2021.

[48] Matt Johnston. Dropbear ssh. `https://github.com/mkj/dropbear`, 2021.

[49] David Kaplan. Protecting VM register state with SEV-ES. *White paper*, 2017.

[50] David Kaplan. Upcoming x86 technologies for malicious hypervisor protection. https://static.sched.com/hosted_files/lsseu2019/65/SEV-SNP%20Slides%20Nov%201%202019.pdf, 2020.

[51] David Kaplan, Jeremy Powell, and Tom Woller. AMD memory encryption. *White paper*, 2016.

[52] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy*, pages 1–19. IEEE, 2019.

[53] Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[54] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Tagbleed: Breaking kaslr on the isolated kernel address space using tagged tlbs. In *2020 IEEE European Symposium on Security and Privacy*, pages 309–321. IEEE, 2020.

[55] Adam Langley. Checking that functions are constant time with valgrind. https://www.imperialviolet.org/2010/04/01/ctgrind.html, 2010.

[56] Byoungyoung Lee, Long Lu, Tielei Wang, Taesoo Kim, and Wenke Lee. From zygote to morula: Fortifying weakened aslr on android. In *2014 IEEE Symposium on Security and Privacy*, pages 424–439. IEEE, 2014.

[57] Dayeol Lee, Dongha Jung, Ian T. Fang, Chia-che Tsai, and Raluca Ada Popa. An off-chip attack on hardware enclaves via the memory bus. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 487–504. USENIX Association, 2020.

[58] Jae-Hyuk Lee, Jin Soo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *26th USENIX Security Symposium*, 2017.

[59] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium*, 2017.

[60] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. CROSSLINE: Breaking "Security-by-Crash" based Memory Isolation in AMD SEV. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2937–2950, 2021.

[61] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. Exploiting unprotected I/O operations in amd's secure encrypted virtualization. In Nadia Heninger and Patrick Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 1257–1272. USENIX Association, 2019.

[62] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. CIPHERLEAKS: breaking constant-time cryptography on AMD SEV via the ciphertext side channel. In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 717–732. USENIX Association, 2021.

[63] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. Take a way: Exploring the security implications of AMD's cache way predictors. In *15th ACM ASIA Conference on Computer and Communications Security (ACM ASIACCS 2020)*, 2020.

[64] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium*, pages 973–990, 2018.

[65] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: cross-cores cache covert channel. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 46–64. Springer, 2015.

[66] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the other side: Ssh over robust cache covert channels in the cloud. In *Proceedings of the Network and Distributed System Security Symposium*, volume 17, pages 8–11, 2017.

[67] Saeid Mofrad, Fengwei Zhang, Shiyong Lu, and Weidong Shi. A comparison study of intel SGX and AMD memory encryption technology. In *7th International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 2018.

[68] Mathias Morbitzer, Manuel Huber, and Julian Horsch. Extracting secrets from encrypted virtual machines. In Gail-Joon Ahn, Bhavani M. Thuraisingham, Murat Kantarcioglu, and Ram Krishnan, editors, *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy, CODASPY 2019, Richardson, TX, USA, March 25-27, 2019*, pages 221–230. ACM, 2019.

[69] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. Severed: Subverting amd's virtual machine encryption. In Angelos Stavrou and Konrad Rieck, editors, *Proceedings of the 11th European Workshop on Systems Security, EuroSec@EuroSys 2018, Porto, Portugal, April 23, 2018*, pages 1:1–1:6. ACM, 2018.

[70] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *Cryptographers' track at the RSA conference*, pages 1–20. Springer, 2006.

[71] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. Make sure DSA signing exponentiations really are constant-time. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1639–1650, 2016.

[72] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: exploiting DRAM addressing for cross-cpu attacks. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 565–581. USENIX Association, 2016.

[73] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. Dude, is my code constant time? In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 1697–1702. IEEE, 2017.

[74] AMD Roger Lai. Amd security and server innovation. *UEFI PlugFest-March*, pages 18–22, 2013.

[75] Brian Rogers, Siddhartha Chhabra, Milos Prvulovic, and Yan Solihin. Using address independent seed encryption and bonsai Merkle trees to make secure processors os- and performance-friendly. In *40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007.

[76] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. *arXiv preprint arXiv:1905.05726*, 2019.

[77] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. *Malware Guard Extension: Using SGX to Conceal Cache Attacks*. Springer International Publishing, 2017.

[78] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. Sgx-shield: Enabling address space layout randomization for SGX programs. In *24th Annual Network and Distributed System Security Symposium*, 2017.

[79] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *14th ACM Conference on Computer and Communications Security*. ACM, 2007.

[80] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing page faults from telling your secrets. In *11th ACM on Asia Conference on Computer and Communications Security*, 2016.

[81] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 317–328, 2016.

[82] Patrick Simmons. Security through amnesia: a software-based solution to the cold boot attack on disk encryption. In Robert H'obbes' Zakon, John P. McDermott, and Michael E. Locasto, editors, *Twenty-Seventh Annual Computer Security Applications Conference, ACSAC 2011, Orlando, FL, USA, 5-9 December 2011*, pages 73–82. ACM, 2011.

[83] Teja Singh, Alex Schaefer, Sundar Rangarajan, Deepesh John, Carson Henrion, Russell Schreiber, Miguel Rodriguez, Stephen Kosonocky, Samuel Naffziger, and Amy Novak. Zen: An energy-efficient high-performance x86 core. *IEEE Journal of Solid-State Circuits*, 53(1):102–114, 2017.

[84] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing analysis of keystrokes and timing attacks on ssh. In *10th USENIX Security Symposium*, 2001.

[85] David Suggs, Mahesh Subramony, and Dan Bouvier. The AMD "Zen 2" processor. *IEEE Micro*, 40(2):45–52, 2020.

[86] Shivam Swami and Kartik Mohanram. COVERT: counter overflow reduction for efficient encryption of non-volatlle memories. In David Atienza and Giorgio Di Natale, editors, *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*, pages 906–909. IEEE, 2017.

[87] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium*, pages 991–1008, 2018.

[88] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *26th USENIX Security Symposium*. USENIX Association, 2017.

[89] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. *2019 IEEE Symposium on Security and Privacy*, 2019.

[90] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In *ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[91] Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. The severest of them all: Inference attacks against secure virtual enclaves. In Steven D. Galbraith, Giovanni Russello, Willy Susilo, Dieter Gollmann, Engin Kirda, and Zhenkai Liang, editors, *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security, AsiaCCS 2019, Auckland, New Zealand, July 09-12, 2019*, pages 73–85. ACM, 2019.

[92] David Weston and Matt Miller. Windows 10 mitigation improvements. *Black Hat USA*, 2016.

[93] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. Sevurity: No security without integrity : Breaking integrity-free memory encryption with minimal assumptions. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1483–1496. IEEE, 2020.

[94] Yuming Wu, Yutao Liu, Ruifeng Liu, Haibo Chen, Binyu Zang, and Haibing Guan. Comprehensive VM protection against untrusted hypervisor through retrofitted AMD memory encryption. In *International Symposium on High Performance Computer Architecture*, 2018.

[95] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy*. IEEE, 2015.

[96] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656. IEEE, 2015.

[97] Chenyu Yan, B. Rogers, D. Englender, D. Solihin, and M. Prvulovic. Improving cost, performance, and security of memory encryption and authentication. In *33rd International Symposium on Computer Architecture*, 2006.

[98] Yuval Yarom and Katrina Falkner. FLUSH+ RELOAD: a high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium*, pages 719–732, 2014.

[99] Min Hong Yun and Lin Zhong. Ginseng: Keeping secrets in registers when you distrust the operating system. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[100] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 305–316, 2012.

[101] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 990–1003, 2014.